A Framework for Integrated Process and Object Life Cycle Modeling

by
Ksenia WAHLER
(born RYNDINA)
from
Russia

DISSERTATION

for the degree of
DOCTOR IN INFORMATICS
at the Faculty of Economics,
Business Administration and
Information Technology
of the University of Zurich

Accepted on the recommendation of
Prof. Dr. Harald C. GALL, University of Zurich (advisor)
Prof. Dr. Schahram DUSTDAR, Vienna University of Technology (co-advisor)

Industrial advisors:
Dr. Jochen M. KÜSTER, IBM Research (advisor)
Dr. Jana KOEHLER, IBM Research (co-advisor)

March 2009

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, March 22, 2009

The Vice Dean of the Academic Program in Informatics: Prof. Dr. Harald C. Gall

Abstract

Modeling has proven to play an essential role in the field of Business Process Management. In particular, process models are created to capture the order in which tasks of a business process are to be executed. Each task typically processes one or more business objects to contribute to the final goal of the business process. A business object can be associated with a number of distinct states to mark the milestones in its overall processing. Many process modeling languages allow one to represent how business objects flow and change their state between tasks.

States of a business object abstract from the details of the performed tasks and are therefore useful in communicating progress to stakeholders who are unaware of the exact process logic. All possible state transitions for a particular type of business object are captured in an object life cycle model. As opposed to a process model, an object life cycle model abstracts from some of the details of the underlying business processes.

Process and object life cycle models hence represent two complementary and overlapping views on the operations of a business. In multi-view modeling approaches, consistency of overlapping models is a critical issue that needs to be addressed. Since the relationship between process and object life cycle models is not yet well-understood, no existing tools or methods offer support for consistency management of such models. As a result, models are created without any means of determining whether they contain inconsistencies, a prime example being the reference models offered in the IBM Insurance Application Architecture[1].

In this dissertation, we develop a framework for integrated process and object life cycle modeling that addresses the challenges of using these models as complementary views. The framework is based on the fundamental constructs of process and object life cycle models and is thus applicable to different modeling languages. As an essential foundation, we provide a formalization of the syntax and semantics for process and object life cycle models. Building on top of this foundation, we define consistency for these model types and describe how it can be checked on a given set of models. We further provide several techniques to assist the modeler in resolving inconsistencies. For the derivation of one view from another, we develop model transformations to extract object life cycle models from process models and to generate process models from object life cycle models.

As tool support for our framework, we implement Object Life Cycle Explorer[2] as an extension to IBM WebSphere Business Modeler[3]. Using this tool support on two industrial case studies, we demonstrate that our approach leads to models of a higher quality.

[1] `http://www.ibm.com/industries/financialservices/doc/content/solution/278918103.html`
[2] `http://www.alphaworks.ibm.com/tech/olcexplorer`
[3] `http://www.ibm.com/software/integration/wbimodeler`

Zusammenfassung

Modellierung spielt erwiesenermassen eine wesentliche Rolle im Gebiet des Prozessmanagements. Insbesondere werden Prozessmodelle benutzt, um die Reihenfolge zu erfassen, in welcher die Tasks eines Geschäftsprozesses auszuführen sind. In jedem Task werden typischerweise ein oder mehrere Businessobjekte bearbeitet, was zur Erreichung der Ziele des Geschäftsprozesses beiträgt. Jedem Businessobjekt kann eine Menge von eindeutigen Zuständen zugewiesen werden, um die Meilensteine in der gesamten Bearbeitung des Objekts festzulegen. In vielen Prozessmodellierungssprachen kann man ausdrücken, wie Businessobjekte durch den Prozess fliessen und zwischen den einzelnen Tasks ihren Zustand ändern.

Zustände von Businessobjekten abstrahieren von den Details der ausgeführten Tasks und sind deshalb nützlich, um den Fortschritt des Prozesses zu kommunizieren, ohne Details der Prozesslogik offenzulegen. Alle möglichen Zustandsübergänge für einen bestimmten Typ von Businessobjekt werden in einem Objektlebenszyklusmodell erfasst. Im Gegensatz zu einem Prozessmodell abstrahiert ein Objektlebenszyklusmodell von bestimmten Details des zugrunde liegenden Geschäftsprozesses.

Prozess- und Objektlebenszyklusmodelle stellen daher zwei sich ergänzende und überlappende Sichten auf die Arbeitsabläufe eines Betriebs dar. In Modellierungsansätzen mit mehreren Sichten auf ein Modell stellt jedoch die Konsistenz von sich überlappenden Modellen ein kritisches Problem dar, auf das eingegangen werden muss. Da die Beziehung zwischen Prozess- und Objektlebenszyklusmodellen noch nicht wohlverstanden ist, bieten die existierenden Werkzeuge und Methoden keine Unterstützung für das Konsistenzmanagement von solchen Modellen. Als Folge davon werden Modelle erstellt ohne Mittel um festzustellen, ob sie Inkonsistenzen enthalten. Ein erstklassiges Beispiel sind die Referenzmodelle, die im Rahmen der IBM Insurance Application Architecture[4] angeboten werden.

In dieser Dissertation wird ein Framework für die integrierte Modellierung von Prozessen und Objektlebenszyklen entwickelt, das auf die Herausforderungen eingeht, wenn diese Modelle als sich ergänzende Sichten benutzt werden. Das Framework basiert auf grundlegenden Konstrukten von Prozess- und Objektlebenszyklusmodellen und ist daher auf verschiedene Modellierungssprachen anwendbar. Als wichtige Grundlage wird eine Formalisierung der Syntax und Semantik von Prozess- und Objektlebenszyklusmodellen erstellt. Auf Basis dieser Grundlage wird die Konsistenz von diesen Modelltypen definiert und beschrieben, wie die Konsistenz einer gegeben Modellmenge überprüft werden kann. Darüberhinaus werden verschiedene Techniken erstellt, die einen Modellierer darin unterstützen, Inkonsistenzen zu beseitigen. Für die Herleitung von einer Sicht aus einer anderen werden Modelltransformationen entwickelt, die Objektlebenszyklusmod-

[4] `http://www.ibm.com/industries/financialservices/doc/content/solution/278918103.html`

elle aus Prozessmodellen extrahieren und Prozessmodelle aus Objektlebenszyklusmodellen generieren.

Die Werkzeugunterstützung für das Framework wird als Object Life Cycle Explorer[5] implementiert, der eine Erweiterung des IBM WebSphere Business Modeler[6] darstellt. Um zu zeigen, dass der vorgestellte Ansatz zu höherwertigen Modellen führt, wird diese Werkzeugunterstützung in zwei industriellen Fallstudien angewandt.

[5] http://www.alphaworks.ibm.com/tech/olcexplorer

[6] http://www.ibm.com/software/integration/wbimodeler

Acknowledgements

In my eyes, the past four years have been a quest, a trial and an adventure, all at once. I would like to thank the following people for seeing me through to the finish line:

My advisor, Prof. Harald Gall, for being so accessible, encouraging and having the ability to always provide criticism with a concrete suggestion for improvement. Harald, from the many meetings we had, I always walked out of your office with a positive attitude and some new ideas.

Prof. Schahram Dustdar, for agreeing to be my co-advisor, making the time for my visit to his lab and finally for reviewing my dissertation.

Jochen Küster, my industrial advisor, for teaching me how to work scientifically and so devotedly supporting me from day one all the way through to my defense. Jochen, thanks for letting me in on your tips and tricks when it comes to scientific writing; I have learnt a great deal from co-authoring papers and patents with you. I appreciate the openness of all our discussions and the time you took for reading my chapters over and over again. I am leaving with some fond and funny memories from our business trips together.

My manager, Jana Koehler, for making it possible for me to work in the Business Integration Technologies group. Jana, thank you for being so responsive with feedback and always pushing me to aim higher. Aside from the research work, I enjoyed the opportunities you gave me to present in front of clients and all the great team-building events that you organized.

Aurelien Monot, for all the hard work he has put into ensuring that our alphaWorks release of Object Life Cycle Explorer went out on time. It was fun working with you, Aurelien!

Martin Harris, for all the valuable feedback on Object Life Cycle Explorer and for making the CAD Management case study possible. Sakae Iwasawa and other members of the IAA development team, for the many insightful discussions about the insurance reference models.

Prof. Pieter Kritzinger, for inspiring me to do a PhD and to apply to the IBM Zurich Research Lab in the first place.

All the current and past members of the Business Integration Technologies group, for creating a stimulating and interesting working environment. A special thanks to Jussi Vanhatalo, for sharing so many ups an downs of the PhD life with me. Jussi, I will really miss sharing an office with you and challenging you in on the pool table. Olaf Zimmermann, thanks for all the entertaining anecdotes you shared with us during lunchtime and for being such great company at music festivals. Cédric Favre, I am glad that we shared our passion for morning coffee!

My mom and dad, for being so present in my life and giving me so much support

despite being on the other side of the world.

My husband, Michael, for never letting me overlook my achievements and always giving me a reason to smile. Thanks for all your patience and good advice.

Contents

Chapter 1

Introduction

In this chapter, we motivate the work detailed in this dissertation and present our research contributions. At the end of the chapter, we provide a structural overview of the dissertation.

1.1 Motivation and Problem Statement

Business Process Management (BPM) is a discipline that encompasses various activities targeted at increasing the effectiveness and efficiency of *business processes* within an organization using information systems [Smith and Fingar, 2006]. A business process is defined as a collection of ordered tasks that need to be carried out in order to achieve a particular business goal. The tasks within a business process can represent automated operations or manual operations performed by humans. BPM activities are usually organized into several phases such as analysis, design, implementation, enactment, monitoring and evaluation [Hollingsworth, 2004, zur Muehlen, 2004, Dumas et al., 2005]. During the analysis, design and implementation phases, a software application is developed to automate the coordination of business process tasks. The performance of the enacted application is observed and evaluated during the monitoring and evaluation phases, based on which the application design is adapted for continuous improvement.

Modeling has proven to be an essential instrument in many of the BPM phases. Most importantly, *process models* are created to capture the business process coordination logic that needs to be automated [Leymann and Roller, 2000, van der Aalst and van Hee, 2004, Dumas et al., 2005]. A process model comprises a set of tasks associated with a *control-flow* relation to indicate the order of task execution. The flow of data or *business objects* between the tasks is also commonly indicated in a process model. For instance, Figure 1.1(a) shows a process model for the handling of insurance claims in the notation of UML Activity Diagrams [UML, 2007b]. $Register\ Claim$, $Evaluate\ Claim$ and $Settle\ Claim$ are examples of tasks in this process model, while $Claim$ is the business object passed between these tasks. The control-flow relation is represented using edges and additional control-flow nodes in this process model. Such high-level process models are usually refined or transformed into an executable *process implementation* [Koehler et al., 2003, Ouyang et al., 2006], which lies at the heart of the overall application developed to manage business processes.

Applications developed in the context of BPM are however multi-faceted, spanning aspects other than the flow of control and business objects. Further aspects include or-

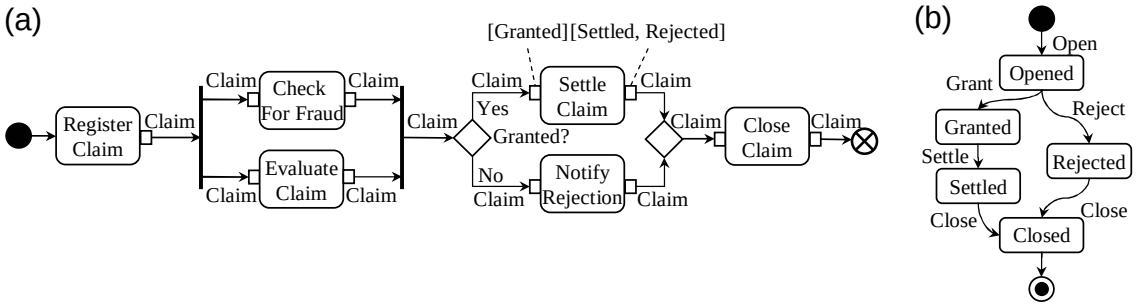


Figure 1.1: (a) Claims handling process model (b) Claim object life cycle model

ganization structure, resource availability, structure and relation of business objects, etc. *Business object state* is another such aspect, which is arguably less prominent in the current literature despite its prevalence in the existing applications. Several books in the BPM area [Mertins and Jochem, 1999, Sharp and McDermott, 2001] concur that business objects are commonly associated with a number of distinct *states*, sometimes also referred to as statuses. A state of a business object corresponds to a milestone in the overall processing of the object, conveying the result of the processing in a manner that abstracts from the details of the operations that were performed. States of business objects change as a result of business process task execution. Once an application is enacted, business object states can be monitored and made available to various stakeholders interested in the attainment of related milestones. For example, by observing the states of all the ongoing insurance claims, a manager can obtain an overview of the progress and performance of the business processes related to claims handling. States that can be associated with an insurance claim among others include $Granted$, $Settled$ and $Rejected$.

Several of the existing process modeling languages allow one to indicate the possible input and output states of tasks by annotating the flow of business objects in a process model [UML, 2007b, BPMN, 2008]. In the example shown in Figure 1.1(a), the $Settle\ Claim$ task is annotated to indicate that it can accept a claim in state $Granted$ and produce a claim in state $Settled$ or $Rejected$. On the other hand, the overall state evolution of business objects can be represented using *object life cycle models* [Kappel and Schrefl, 1991, Ebert and Engels, 1997, Date, 2000, Stumptner and Schrefl, 2000]. An object life cycle model defines the possible states and transitions between these states for objects of a particular object type. Distinguished initial and final states are used to mark the beginning and end points of object processing in an object life cycle model. For example, Figure 1.1(b) shows an object life cycle model for insurance claims. In accordance with this object life cycle model, an insurance claim object can transition from state $Granted$ to $Settled$, but not from $Rejected$ to $Settled$. In the context of BPM, object life cycle models can be used for specifying state evolution requirements and design for business objects during the analysis and design phases, and for interpreting business object states observed during the enactment of an application. Furthermore, some more recent approaches to process implementation make direct use of object life cycle models for coordinating business process tasks during enactment [van der Aalst et al., 2001, Nandi and Kumaran, 2005, Müller et al., 2006].

Process and object life cycle models hence represent two complementary views on an application: the *process view* focusing on the flow of control and business objects between tasks and the *object life cycle view* capturing the overall state evolution of business objects. Since the states of business objects are changed due to the execution of

tasks in a process model, these two types of models clearly represent overlapping behavior. In multi-view approaches to software development, it has been shown that *consistency* is a critical issue that needs to be addressed when using such overlapping models [Finkelstein et al., 1992, Grundy et al., 1998, Nuseibeh et al., 2001]. A set of models can only be considered consistent if they do not contain contradicting assertions about the application being developed. For example, the process and object life cycle models shown in Figure 1.1 should not be considered consistent, since the claims handling process model can change the state of a claim from $Granted$ to $Rejected$, while this state transition is not defined in the object life cycle model for claims.

Inconsistent models make it impossible to implement an application that satisfies all of the requirements specified in these models. In the case where the implemented application only satisfies some of the requirements captured in the models, the models can no longer be used as a basis for rationalizing about the behavior of the application. In the example of insurance claims handling, incorrect interpretation of claim states by stakeholders such as managers and clients can lead to inappropriate decision-making. For instance, a client who expects that a $Granted$ claim is always $Settled$ according to the object life cycle model in Figure 1.1(b) could decide to take out a loan assuming to later repay it using the claim settlement. This decision can leave the client in a predicament if the claim later gets $Rejected$ according to the implementation of the claims handling process model shown in Figure 1.1(a).

Determining and managing consistency between process and object life cycle models is not trivial. In practice, process models are never as simple as the example shown in Figure 1.1(a). A real process model can comprise tens or even hundreds of tasks structured into hierarchy levels, intricate control flows involving concurrency and intertwined flows of various business objects. Examining such process models manually with respect to state changes they can induce on different business objects is infeasible. Furthermore, the semantics of many process modeling languages such as Event-Driven Process Chains (EPCs) [Keller et al., 1992], UML Activity Diagrams [UML, 2007a] and Business Process Modeling Notation (BPMN) [BPMN, 2008], is not precisely defined, especially with respect to the flow of business objects (commonly referred to as *data flow*). This challenges the definition of consistency even if simple process models are considered.

At present, management of consistency between process and object life cycle models is not addressed by any of the existing modeling languages, tools or methods. The de facto standard for software modeling, UML [UML, 2007b], can be used to model processes and object life cycles using UML Activity Diagrams and UML State Machines, respectively. However, the UML specification does not explain the relationship between these models or define any constraints between their elements. Integration of EPCs with object-oriented models, including object life cycle models, has been described on a very high level in [Loos and Allweyer, 1998, Loos and Fettke, 2001]. However, the authors highlight the benefits of and provide a motivation for integrating these models rather than a complete solution. In [Redding et al., 2007], a translation of object life cycle models in a custom representation to process models in Yet Another Workflow Language (YAWL) [van der Aalst and ter Hofstede, 2005] is described. Although a particular relation between these models is assumed for the translation, this relationship is not made explicit and the consistency between the models is not addressed.

In this dissertation, we are concerned with clarifying the relationship between process and object life cycle models and addressing the fundamental challenges of using these types of models as complementary views in the context of BPM. An overview of our proposed solution and an account of the research contributions are provided in the following

section.

1.2 Proposed Solution and Research Contributions

As a solution for integrating process and object life cycle modeling, we propose a framework comprising several components as shown in Figure 1.2. Components 1-4 address the issues that arise from the integration of process and object life cycle modeling during the analysis and design phases of BPM. As part of the framework, we also explore how to leverage object life cycle models during the transition from the design phase to the implementation phase (see component 5). Tool support is implemented for the entire framework to demonstrate the feasibility of this solution and to enable its validation.

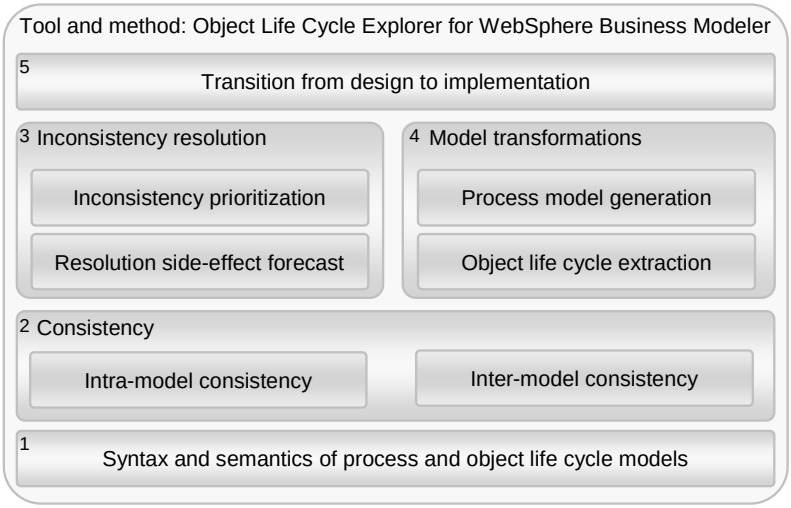


Figure 1.2: Proposed framework for integrated process and object life cycle modeling

In the following, we describe each of these components and point out their underlying research contributions.

1. **Syntax and semantics of process and object life cycle models:** As a basis for establishing the relationship between process and object life cycle models and defining their consistency, we provide a precise definition of their syntax and semantics. For process models, we extend an existing syntactic and semantic definition of control-flow-only process models [Vanhatalo et al., 2007] with data flow and object states. We cover two main approaches to representing data flow in process models (direct routing between nodes vs. routing via intermediary repositories) and two main approaches to passing data between tasks (by value vs. by reference), distinguished in the existing literature [Sadiq et al., 2004, Russell et al., 2005]. For object life cycle models, we tailor the definition and interpretation of finite state automata [Hopcroft et al., 2006] for using these models as protocols of object state evolution. The main contributions are as follows:

   - A formalization of the *syntax* for data flow and object states in process models, incorporating data flow that is either *routed directly* or via *repositories*;
   - A definition of the *semantics* for process models with data flow and object states that distinguishes the *pass-by-value* and *pass-by-reference* approaches;

- A definition of object life cycle *conformance* and *coverage* that captures the *semantics* of object life cycle models as state evolution protocols.

2. **Consistency:** We address intra-model consistency, which is consistency of elements in one model, as a prerequisite for defining inter-model consistency between process and object life cycle models. In this regard, we define the correctness of process models with object states, unaddressed so far in the state-of-the-art literature. For object life cycle models, we do not identify elaborate intra-model consistency issues. To address inter-model consistency between process and object life cycle models, we map the models to a common semantic domain of object state sequences and use it to define the consistency. For the evaluation of correctness of process models with object states and consistency of process and object life cycle models, we define sets of syntactic conditions and show how they can be decided using a static analysis technique based on iterative data-flow analysis [Kam and Ullman, 1976]. The main contributions are as follows:

   - A definition of *correctness* for process models with object states;
   - A set of *syntactic conditions* that are necessary and sufficient with regards to the above-mentioned correctness definition for a particular class of process models, and can be statically evaluated over a given process model;
   - A definition of *consistency* for process and object life cycle models;
   - A set of *syntactic conditions* that are necessary and sufficient with regards to the above-mentioned consistency definition for a particular class of process models, and can be statically evaluated over given process and object life cycle models.

3. **Inconsistency resolution:** Checking consistency between process and object life cycle models can result in inconsistencies being detected. In accordance with existing literature [Van Der Straeten, 2005], resolution of a model inconsistency can be facilitated by implementing a model transformation that automatically changes elements of the inconsistent models to remove the inconsistency. Since changing some model elements to resolve one inconsistency can introduce or remove other inconsistencies, inconsistency resolutions in process and object life cycle models may have side-effects and may even lead to cycles in the resolution process [Mens et al., 2006b, Egyed, 2007]. We propose an approach to developing inconsistency resolutions with explicit specification of so-called side-effect expressions, which facilitates the forecast of side-effects for a given resolution and whether or not it can lead to a cycle. This information makes it easier for the modeler to choose among alternative resolutions for the same inconsistency. To alleviate the resolution of large sets of inconsistencies, we additionally provide a method for inconsistency prioritization to guide the modeler through a set of inconsistencies in a way that minimizes the overall time and effort required to switch between the different inconsistencies. The main contributions, which also apply outside the context of process and object life cycle models, are as follows:

   - A method for *inconsistency prioritization* that minimizes the overall time and effort required by the modeler to switch between the different inconsistencies;
   - An approach to developing inconsistency resolutions that facilitates the *forecast* of *resolution side-effects*;

- An analysis technique for determining whether an inconsistency resolution can lead to a *resolution cycle*.

4. **Model transformations:** In pursuit of a seamless integration of process and object life cycle models as complementary views, we define transformations from process models to object life cycle models and vice versa. We provide the object life cycle extraction transformation that generates an object life cycle model for each object type manipulated in a given process model. Analogously, we provide the process generation transformation to construct a process model from a set of given object life cycle models. These two transformations ensure the consistency of the produced models with the input models. We additionally define minimality criteria for process and object life cycle models to show that the models generated by the transformations are minimal in the set of all models consistent with the input models. By showing these properties, we go beyond the existing work on transformations from object life cycle to process models [Redding et al., 2007]. The main contributions are as follows:

   - A *transformation* from process to object life cycle models, which ensures *consistency* and *minimality* of the produced object life cycle models;
   - A *transformation* from a set of object life cycle models to a process model, which ensures *consistency* and *minimality* of the produced process model.

5. **Transition from design to implementation:** In the transition from design to implementation, we especially consider leveraging object life cycle models in the so-called object-centric approaches to process implementation [van der Aalst et al., 2001, Nandi and Kumaran, 2005, Müller et al., 2006]. These approaches distribute process control-flow logic among several interacting components, each associated with an object life cycle model. One of the challenges is the management of interdependencies or coupling between such components, since high coupling makes it difficult to distribute, maintain and adapt the components. Our object life cycle extraction transformation does not necessarily preserve all the control-flow logic of a process model, since its main purpose is the generation of a complementary view. Therefore, we firstly investigate how a complete decomposition of control flow captured in a process model among several object life cycle models can be achieved. Based on an informal description of this decomposition, we propose a technique for statically analyzing a given process model to compute the expected object life cycle coupling. This information enables the direct adaptation of the process model to alleviate the coupling of the resulting object life cycle models. The main contributions are as follows:

   - An informal description of a *fully behavior-preserving transformation* from process to object life cycle models;
   - A technique for computing the expected *coupling* of object life cycle models based on a given process model.

Most of these contributions have been published as peer-reviewed conference and workshop papers as illustrated in Figure 1.3. The first version of the transformation from process to object life cycle models and the initial ideas on consistency were published in

the proceedings of the *1st International Workshop on Quality in Modeling (QiM'06)* co-located with *the ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)* [Ryndina et al., 2006]. The consistency definition and conditions were refined in a follow-up publication in the proceedings of the *5th International Conference on Business Process Management (BPM'07)* [Küster et al., 2007], which also presents a technique for transforming object life cycle models to process models. The core of the inconsistency resolution approach, including the forecast of resolution side-effects, was published in the proceedings of *MoDELS'07* [Küster and Ryndina, 2007]. The technique for the computation of the expected coupling in object-centric process implementations was published in the proceedings of *BPM'08* [Wahler and Küster, 2008].

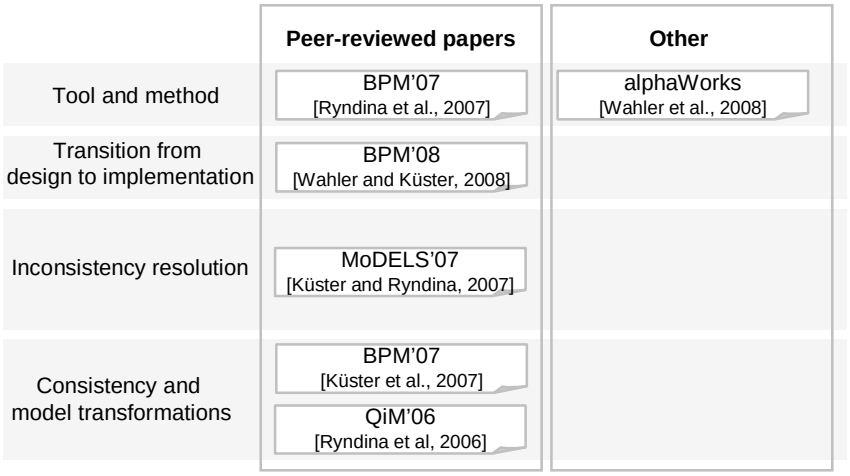


Figure 1.3: Publications overview

The proposed framework is implemented in a prototype called *Object Life Cycle Explorer* and is supported by a method that comprises several modeling strategies to address the main usage scenarios for the framework. Object Life Cycle Explorer is developed as an extension to IBM WebSphere Business Modeler[1], a commercial tool for the analysis and design of process models. Object Life Cycle Explorer was presented in a demo session at *BPM'07* [Ryndina et al., 2007], where it was selected for the best demo award by a panel of leading practitioners. On May 29 2008, Object Life Cycle Explorer was released on *IBM alphaWorks* [Wahler et al., 2008], where it can be freely downloaded for non-commercial purposes. Apart from the software, the download package also contains extensive tutorials and sample models. To this date, the package was downloaded over 100 times and is being used on several client engagements.

Feasibility, effectiveness and added value of our solution were validated using two industrial case studies where explicit modeling of object state evolution was required. Both case studies demonstrated that integrated process and object life cycle modeling according to our solution has significant benefits when compared to state-of-the-art approaches.

The remainder of this dissertation describes the details of the proposed framework, presents the tool and method support, and describes the performed case studies and their results. In the following section, we give an overview of the dissertation structure.

[1] `http://www.ibm.com/software/integration/wbimodeler`

1.3 Dissertation Structure

In Chapter 2, we present the necessary background for the remainder of the dissertation and give a general overview of related work. References to related work additionally appear in some later chapters to provide a more detailed comparison of our solution to the state of the art in respect to some finer details.

Chapter 3 is concerned with the syntax and semantics of process and object life cycle models, presenting definitions used as a foundation in the remainder of the dissertation. In Chapter 4, we address the correctness of process models with object states and the consistency of process and object life cycle models, providing definitions and evaluation techniques for both. In turn, Chapter 5 is concerned with the resolution of inconsistencies that may be identified during consistency checking. In this chapter, we present our proposed approach to inconsistency resolution based on side-effect expressions and describe a method for guiding the modeler through a set of prioritized inconsistencies. To complete our solution for integrating process and object life cycle modeling during the analysis and design phases, we present the transformations from process to object life cycle models and vice versa in Chapter 6.

In Chapter 7, we explore the transition from the design phase to the implementation phase with a special focus on deriving object-centric process implementations. In this chapter, we propose a technique for the computation of the expected coupling of object life cycle models based on a given process model.

Chapter 8 comprises a description of Object Life Cycle Explorer and several proposed modeling strategies that form the method support for our solution. In Chapter 9, we describe the two case studies that we used to validate our solution and discuss the obtained results. Finally, we summarize the contributions made in this dissertation and provide an outlook on possible future research in Chapter 10.

Chapter 2

Background and Related Work

In this chapter, we present the necessary background for the remainder of the dissertation and discuss related work. We begin by a general introduction to how models are used in software engineering in Section 2.1, after which we concentrate on process and object life cycle modeling in Sections 2.2 and 2.3, respectively. For each, we provide an overview of the main modeling concepts and languages. We additionally take a closer look at the use of UML Activity Diagrams and UML State Machines for modeling processes and object life cycles, respectively. As this dissertation is concerned with the integration of process and object life cycle models as complementary views, we provide the background and related work on multi-view modeling in Section 2.4. As three main topics of multi-view modeling, we discuss consistency management, inconsistency management and model transformations. Finally, we discuss the transition from the design phase to the implementation phase in the context of Business Process Management in Section 2.5.

2.1 Models in Software Engineering

Many software engineering methodologies and tools make use of modeling during some or all software development phases [Mayer et al., 1992, Ambler and Jeffries, 2002, Kruchten, 2004]. In this context, a *model* is defined either as an abstract representation of the real world (e.g. an organizational model) or an abstract representation of the application being developed (e.g. an architecture model). Models used for software engineering commonly have a graphical representation, which is considered to make them more accessible for the different stakeholders such as developers and domain experts.

Graphical modeling techniques have been used for the design of software already in the 1970's, the most prominent example being the entity-relationship models used for database design [Chen, 1976]. The concept of having several models for representing different perspectives on the same application became popular with the structured analysis and design techniques such as the Structured Systems Analysis and Design Method (SSADM) [Ashworth and Goodland, 1989]. SSADM distinguishes between logical data modeling, data-flow modeling and entity behavior modeling. The set of models created according to SSADM are collectively used as a specification for the implementation of an application. In the 1990's, object-oriented analysis and design methods [Shlaer and Mellor, 1988, Coad and Yourdon, 1991, Booch, 1994] became increasingly popular, which eventually led to the standardization of the Unified Modeling Language (UML) [UML, 2007b]. UML comprises an integrated set of languages for structural

and behavioral software modeling at different levels of abstraction.

For a significant period of time, the primary usage of models in software engineering was communication among developers. Models were produced to capture the intent of developers in one software development phase and passed on to other developers as a specification for the next phase. The relationships between different models were not well-defined, and the correctness and accuracy of how a specification model was used as a basis for creating another model or code relied largely on the experience of the developers. However, the recent paradigm of Model-Driven Engineering (MDE) [Kent, 2002, Kleppe et al., 2003, Schmidt, 2006], which includes the Model-Driven Architecture (MDA) approach [MDA, 2003] from the Object Management Group, places models in the center of all the software development activities and makes an explicit distinction between the different modeling abstraction levels. According to MDA, a *Platform Independent Model (PIM)* is first created to capture an application at a technology-neutral level and is later automatically translated to a *Platform Specific Model (PSM)*, which can be directly executed on a particular technological platform. With such treatment of models, an unambiguous specification of the syntax and semantics of modeling languages became more important.

Today, the de facto approach to specifying the syntax of a graphical modeling language is using a *meta-model*. A meta-model is a model that defines the modeling elements of a language and the relationships between these elements. Meta-modeling is standardized in Meta-Object Facility (MOF) [MOF, 2002] by the Object Management Group. In addition to a meta-model, textual constraints are sometimes required to further restrict the relationships between model elements. Such constraints can be formalized using Object Constraint Language (OCL) [OCL, 2003]. A MOF meta-model and its accompanying OCL constraints constitute a precise specification of the syntax of a modeling language. On the other hand, the semantics of most modeling languages is only described informally, since there is currently no standard approach to precisely specifying the semantics of a modeling language.

Since this dissertation is focused on process and object life cycle models, we describe these two types of models in detail in the following two sections.

2.2 Process Modeling

While the origins of process modeling can be traced all the way back to work management theories of Taylor and Gantt [Taylor, 1911, Gantt, 1919], the advent of process modeling for automation using software took place in the 1990's marked by the formation of the Workflow Management Coalition[1]. Many process modeling languages and representations have been developed since then. In the following, we present an overview of the most influential ones out of these.

2.2.1 Overview of Process Modeling Languages

Arguably, the most popular languages for process modeling are Event-Driven Process Chains (EPCs) [Keller et al., 1992], UML Activity Diagrams [UML, 2007b] and Business Process Modeling Notation (BPMN) [BPMN, 2008], given here in the chronological order of their development.

[1] `http://www.wfmc.org`

EPCs represent processes as a set of alternating *events* and *functions*. An event represents a significant change of state, which can trigger the start of one or more tasks represented by functions. In an EPC diagram, events and functions are connected by control-flow edges and *logical connectors* such as XOR and AND (see example in Figure 2.1(a)). In addition, functions can be connected to *data objects*[2] and organization units. The use of EPCs was popularized by the ARIS Toolset from IDS Scheer AG [Scheer, 2000].

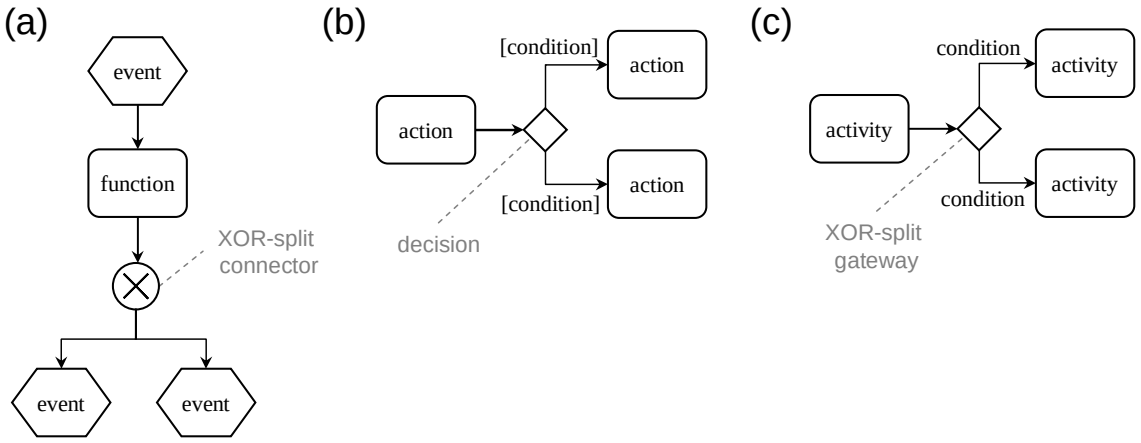


Figure 2.1: Modeling elements in (a) EPCs (b) UML Activity Diagrams (c) BPMN

In UML Activity Diagrams, processes are modeled as a set of *actions* connected by control-flow and data-flow edges. *Control nodes* such as decision, merge, fork and join, are used to model alternative and parallel splits and joins of the process flows (see example in Figure 2.1(b)). UML Activity Diagrams is a very rich language, supporting hierarchical process modeling and various types of data-flow modeling.

BPMN is a graphical notation for modeling business processes, developed with the goal of providing a standard notation that is understandable by a non-technical business stakeholder. Tasks are represented by *activities*, which are connected with sequence flows and *gateways* such as exclusive (XOR) and parallel (AND) splits and joins to represent the flow of control (see example in Figure 2.1(c)). *Events* influencing the flow of control in a business process can be modeled explicitly using start, intermediate and end events. Data flow can be captured by attaching *data objects* to activities and edges.

Table 2.1 gives an overview of how the fundamental process modeling elements are represented in the three languages.

Table 2.1: Fundamental process modeling elements in different languages

Element	**EPCs**	**UML Activity Diagrams**	**BPMN**
task	function	action	activity
control flow	event and arcs	activity edge	sequence flow
data	data object	object node	data object
exclusive choice	XOR-split connector	decision node	XOR-split gateway
simple merge	XOR-join connector	merge node	XOR-join gateway
parallel split	AND-split connector	fork node	AND-split gateway
synchronization	AND-join connector	join node	AND-join gateway

In Table 2.1, apart from “task”, “control flow” and “data”, we also use the terminol-

[2]In EPCs, “data objects” are also sometimes referred to as “information objects”.

ogy introduced by the *workflow patterns* [van der Aalst et al., 2003, Russell et al., 2006a] to describe the main four types of control nodes. The *exclusive choice* corresponds to a control node that splits one control-flow path into many, selecting exactly one of the alternative outgoing paths during execution. Conversely, the *simple merge* joins exclusive control-flow paths into one. The *parallel split* has many outgoing control-flow paths, all of which are enabled concurrently during the execution. Finally, the *synchronization* joins concurrent control-flow paths into one, waiting for each path to complete before enabling the outgoing one. The interested reader is directed to the workflow patterns web site[3] for a much more detailed comparison of the EPCs, UML Activity Diagrams and BPMN.

As can be seen in Table 2.1, data flow can be represented in all three languages. Each language also supports the specification of object states, as illustrated with examples in Figure 2.2. EPCs do not provide a separate construct for object state, but events are commonly expressed as an object and its state, as shown in the first diagram in (a). As suggested in [Loos and Allweyer, 1998], states can alternatively be specified in the object constructs, illustrated in the second diagram in (a). UML Activity Diagrams provide an explicit modeling element for representing object states. One or more states can be attached to object nodes, as illustrated in (b). BPMN also supports explicit modeling of states for data objects, as show in (c). However, only one state can be assigned per data object.

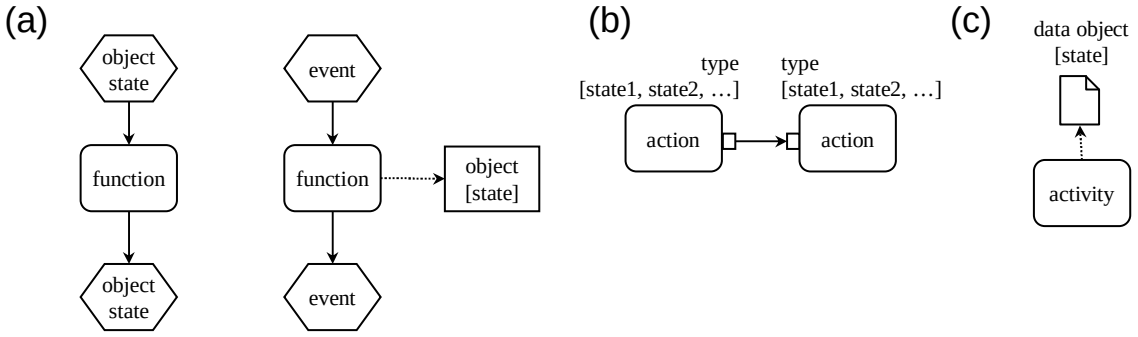


Figure 2.2: Object state modeling in (a) EPCs (b) UML Activity Diagrams (c) BPMN

Out of EPCs, UML Activity Diagrams (V2.1.2) and BPMN (V1.1), UML Activity Diagrams is currently the language that is described most extensively in its *native specification*. The UML Activity Diagrams specification [UML, 2007a] describes the syntax using a meta-model and OCL constraints and the semantics using natural language. In contrast, the syntax and semantics of EPCs are both described only informally. Since their advent, however, several research efforts have been undertaken to formalize EPCs (e.g [van der Aalst, 1999, Kindler, 2006, Mendling, 2007]). At the time of writing, the BPMN specification does not provide a meta-model or a detailed semantic description for the notation, however this is expected to change in BPMN V2.0 [BPMN, 2007]. Several formalizations of the BPMN semantics have also been proposed in the research community [Dijkman et al., 2008, Wong and Gibbons, 2008].

Other existing languages for process modeling include Yet Another Workflow Language (YAWL) [van der Aalst and ter Hofstede, 2005] and XML Process Definition Language (XPDL) [XPDL, 2008]. YAWL was developed to directly support the workflow patterns, after these have been used to identify deficiencies in other existing process modeling languages [Dumas and ter Hofstede, 2001, Wohed et al., 2005, Wohed et al., 2006]. XPDL does not have a graphical representation of its own, but rather defines a standard

[3] `http://www.workflowpatterns.com`

format for the exchange of process models. For instance, it can be used for the exchange of process models in BPMN between different tools.

In this dissertation, we consider the fundamental constructs of process models that occur in most process modeling languages to ensure the general applicability of our results. As the graphical notation, we choose the representation suggested in the UML Activity Diagrams specification. Additionally, we adopt the terminology and some key semantic aspects of UML Activity Diagrams rather than those of EPCs or BPMN. For this reason, we now take a closer look at UML Activity Diagrams.

2.2.2 A Closer Look at UML Activity Diagrams

The core subset of the UML Activity Diagrams meta-model is depicted in Figure 2.3. Italics indicate abstract classes that cannot be instantiated as an element in a concrete model.

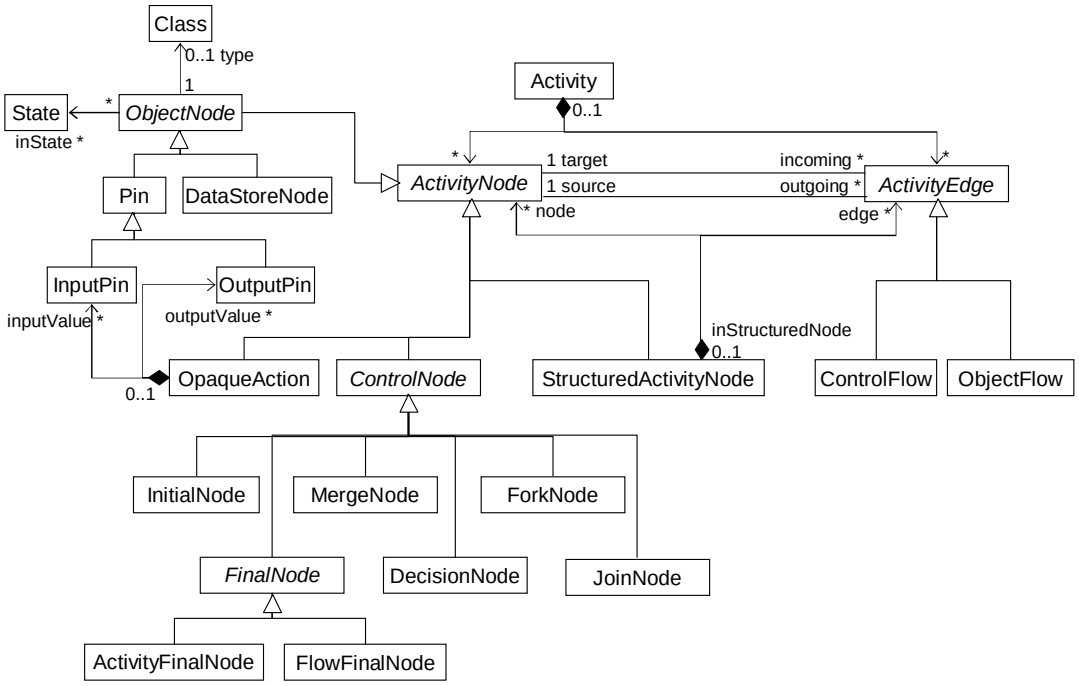


Figure 2.3: Subset of UML Activity Diagram meta-model (V2.1.2) [UML, 2007a]

In UML Activity Diagrams, a process is represented by an Activity, which contains *ActivityNodes* and *ActivityEdges* (cf. Figure 2.3). An Activity that models a claims handling process introduced in Chapter 1 is shown in Figure 2.4 with the different model elements labeled. The functional elements of the process that correspond to tasks to be performed are represented by OpaqueActions[4], e.g. $Register\ Claim$ and $Evaluate\ Claim$. An OpaqueAction is a black-box, since it abstracts from the implementation details of the task at hand. UML Activity Diagrams also distinguishes between other types of actions, e.g. AcceptEventAction and CreateObjectAction, which we do not consider here.

ControlNodes are used to model the control-flow logic of a process, which defines when the different actions should be performed. An InitialNode indicates the start point of the process, and the *FinalNodes* indicate the end points of the process. A FlowFinalNode indicates the end of its incoming flows only, while an ActivityFinalNode indicates an immediate termination of the entire process. Alternative flow branching and merging are represented

[4]OpaqueAction is a specialization of the abstract class Action in the complete meta-model.

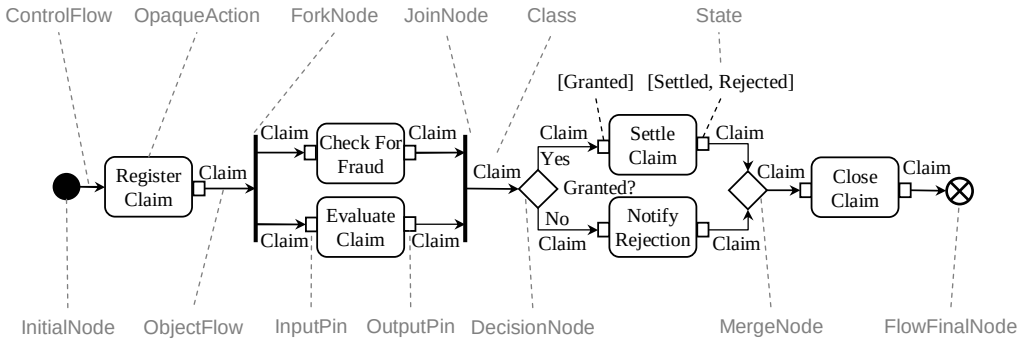


Figure 2.4: Activity with directly routed data flow

using DecisionNodes and MergeNodes, respectively. ForkNodes and JoinNodes are used to model parallel flow branching and merging, respectively. OpaqueActions and ControlNodes are connected with ControlFlows to represent the complete control flow logic of a process.

Data flow can be modeled in several ways in UML Activity Diagrams: data objects can be passed directly from one *ActivityNode* to another, OpaqueActions can read and write objects to and from intermediate data repositories, or a combination of the two approaches can be used. Regardless of which approach is used, ObjectFlows are used to model the channels along which data objects are passed. For direct object routing OpaqueActions should be associated with InputPins that show which data objects they require as an input, and OutputPins that show which data objects they produce as an output. The routing of an object is then modeled by connecting an OutputPin of an OpaqueAction to an InputPin of another OpaqueAction with ObjectFlows, possibly with several intermediate *ControlNodes*. This is the approach used in the Activity shown in Figure 2.4.

To use the approach with intermediate repositories, OpaqueActions should be connected with DataStoreNodes using ObjectFlows to represent the reading and writing of objects, as shown in Figure 2.5. In the graphical representation suggested in the UML specification, DataStoreNodes are connected directly to OpaqueActions with ObjectFlows. However, because this clutters the diagram, we introduce a custom notation where the ObjectFlows are split in two and their end-points indicate the direction and the name of the DataStoreNode to which they should be connected. For example, in Figure 2.5, $Register\ Claim$ only writes objects to the $Claim$ DataStoreNode, while $Evaluate\ Claim$ also reads objects from this DataStoreNode. *Pins* and DataStoreNodes are special types of *ObjectNodes*, and each *ObjectNode* is associated with a Class to indicate the type of objects it can contain (cf. meta-model in Figure 2.3).

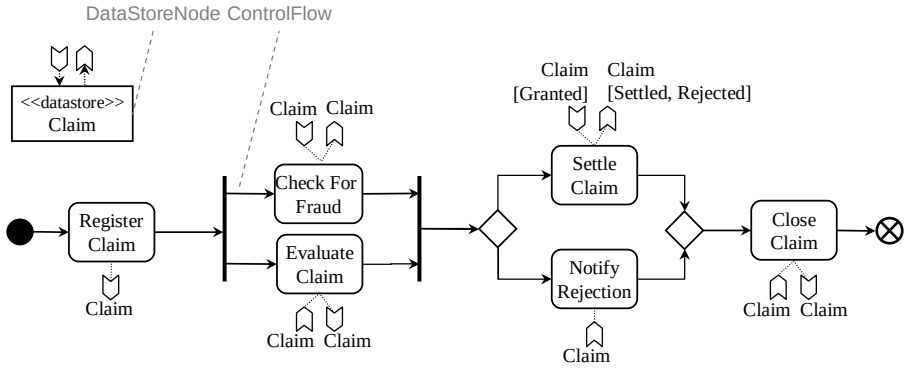


Figure 2.5: Activity with data flow via repositories

In the meta-model extract in Figure 2.3, it is also shown that an *ObjectNode* can be associated with several States to model the states of an object at a particular point in a process. For example, the InputPin of $Settle\ Claim$ is associated with state $Granted$ in Figure 2.4 to indicate that this action requires $Claim$ objects received on this pin to be in state $Granted$. In Figure 2.5, states are shown for the incoming and outgoing object flows connecting $Settle\ Claim$ to the $Claim$ DataStoreNode. The states are actually associated with pins of the $Settle\ Claim$, which are elided from the diagram.

Hierarchical modeling of processes is facilitated using StructuredActivityNodes, which represent subprocesses of the main process. A StructuredActivityNode is connected to other nodes in an Activity as any other *ActivityNode*, but in contrast to OpaqueActions, it has an internal structure and itself contains other *ActivityNodes* and *ActivityEdges*.

The execution semantics of the different *ActivityNodes* is described based on the *token* concept, originating from Petri nets [Peterson, 1981, Murata, 1989]. According to the UML Activity Diagram specification [UML, 2007a, pg. 318], a token "contains an object, datum, or locus of control" and during the execution it is "is present in the activity diagram at a particular node". For each type of ActivityNode, rules governing its consumption and production of tokens are described. For example, each token arriving at a DecisionNode is routed to one of the outgoing edges, while each token arriving at a ForkNode is duplicated across all outgoing edges. The semantics of data flow is captured by *object tokens*, which are associated with a type and are therefore distinguished from control tokens.

Although the semantics descriptions in UML V2.x specifications are extensive, they have been significantly criticized for being imprecise and ambiguous. Bock has written a series of articles to explain and discuss different aspects of the UML Activity Diagrams semantics [Bock, 2003a, Bock, 2003b, Bock, 2003c, Bock, 2004]. Semantic ambiguities of data flow and so-called interrupt regions in UML V2.0 are discussed by Eshuis in his PhD dissertation [Eshuis, 2002], which is primarily concerned with the formalization of the semantics of Activity Diagrams in UML V1.4. Different approaches to formalizing the semantics of UML Activity Diagrams (V2.x) can be found in the existing literature. For example, Stoerrle formalizes the main control-flow semantics using Petri nets [Stoerrle, 2004] and some aspects of the data-flow semantics using colored Petri nets (CPN) [Stoerrle, 2005]. Vitolins and Kalnins describe a virtual machine that uses so-called push and pull token engines for the execution of UML Activity Diagrams [Vitolins and Kalnins, 2005]. Sarstedt and Guttmann formalize the semantics of UML Activity Diagrams using Abstract State Machines [Sarstedt and Guttmann, 2007]. Despite the clarifications of the semantics presented in these works, the semantic descriptions in the UML Activity Diagrams specification have not yet been revised to alleviate the ambiguities.

2.3 Object Life Cycle Modeling

Almost all objects or things, tangible or even intangible, usually go through several phases during their life time. For example, the phases of a person's life include infancy, childhood, adolescence, adulthood and old age. The transitions between the phases or states of an object are usually restricted, so that the object cannot just transit from any state to any other state. A person's life cycle is completely deterministic: from infancy there is a transition to childhood, from childhood to adolescence, and so forth. Other examples of life cycles are not deterministic, and may include branching and cycles. For instance, an insurance claim is first $Opened$ and then it either transits to state $Granted$ or to state $Rejected$. Sometimes claimants may appeal against a rejected claim, in which case a claim may be re-opened, and thus transition to state $Opened$ once again.

Several languages and representations are used in the existing literature for modeling object life cycles, as described next.

2.3.1 Overview of Object Life Cycle Modeling Languages

Object life cycle models are most commonly represented using languages based on either finite state automata, finite state machines or statecharts [Ebert and Engels, 1997, Stumptner and Schrefl, 2000], or various types of Petri nets [Kappel and Schrefl, 1991, van der Aalst and Basten, 2001, Schrefl and Stumptner, 2002]. The Jackson System Development method [Jackson, 1983] takes another approach to modeling object life cycles using entity life history diagrams that capture the states and events related to an object.

A finite state automaton or machine [Moore, 1956, Hopcroft et al., 2006] represents an object life cycle as a set of *states* and a directed *transition relationship* between these states. Each transition is generally associated with an *event* triggering the transition. The use of the *event-condition-action* paradigm is also widespread among the state machine languages: once an event occurs and triggers a transition, the condition associated with the transition is evaluated and if it evaluates to true, then the associated action is performed and only then the target state of the transition is entered. One initial and several final states are usually distinguished in a state machine.

Statecharts [Harel and Politi, 1998] extend state machines with hierarchical state modeling. Hierarchical state structure introduces several abstraction levels into a state machine. Whereas state machines are sequential in nature, statecharts can represent concurrency using the so-called orthogonal states. UML State Machines [UML, 2007b] are based on statecharts. The use of UML State Machines for object life cycle modeling has been described in [Stumptner and Schrefl, 2000].

Petri nets [Peterson, 1981, Murata, 1989] is a language for modeling discrete concurrent systems. A Petri net comprises a set of *places*, a set of *transitions* and a set of directed arcs connecting the places and transitions. The places in a Petri net contain tokens that jointly describe the overall state of the system at a particular point in time. The system changes the state by the firing of transitions, which results in tokens moving between places. For the modeling of object life cycles, places correspond to object states and transitions represent the operations that change the state of the object. Object behavior diagrams, based on Petri nets, are introduced in [Kappel and Schrefl, 1991] for object life cycle modeling. In [van der Aalst and Basten, 2001], another variant of Petri nets, called labeled Place/Transition nets, is used for representing object life cycles.

An entity life history diagram models an object life cycle as a tree: the object type is the root of the tree, which has high-level phases of the object's life as its immediate children. Events relevant for a particular object are connected to its phases in the tree. Each event is in turn connected to states, to which the object can transition as a result of the event. Different types of events are distinguished, e.g. sequential, iteration, alternative. A meta-model for entity life histories is presented in [Hay, 2006], but the semantics of these models is not precisely described in the literature.

In this dissertation, we consider object life cycle modeling in the context of BPM. Business objects manipulated by business processes, such as insurance claims, purchase orders, job applications and inventory items, are typically associated with one state at a particular point in time. For example, a claim is either $Opened$, $Granted$ or $Rejected$ at a given point in time, but is not in a combination of these states. Whereas one can think of examples that associate multiple concurrent states for one object, we scope our focus around sequential object life cycles in this work. As a graphical notation for object life

cycles, we use UML State Machines. In the following, we take a closer look at the subset of UML State Machines that can be used for modeling sequential object life cycles.

2.3.2 A Closer Look at UML State Machines

Figure 2.6 shows the core subset of the UML State Machines meta-model. UML distinguishes two types of state machines: *behavioral state machines* and *protocol state machines*. A behavioral state machine (represented by StateMachine in the meta-model) can be used to model the complete behavior of a particular element, whereas a protocol state machine (ProtocolStateMachine in the meta-model) captures a usage protocol of an element abstracting from the implementation. A Transition in a behavioral state machine follows the event-condition-action paradigm and is associated with the following: a trigger that represents the event that triggers the transition, a guard that captures the condition to be evaluated when the transition has been triggered and an effect that corresponds to an action that is performed if the guard is evaluated to true.

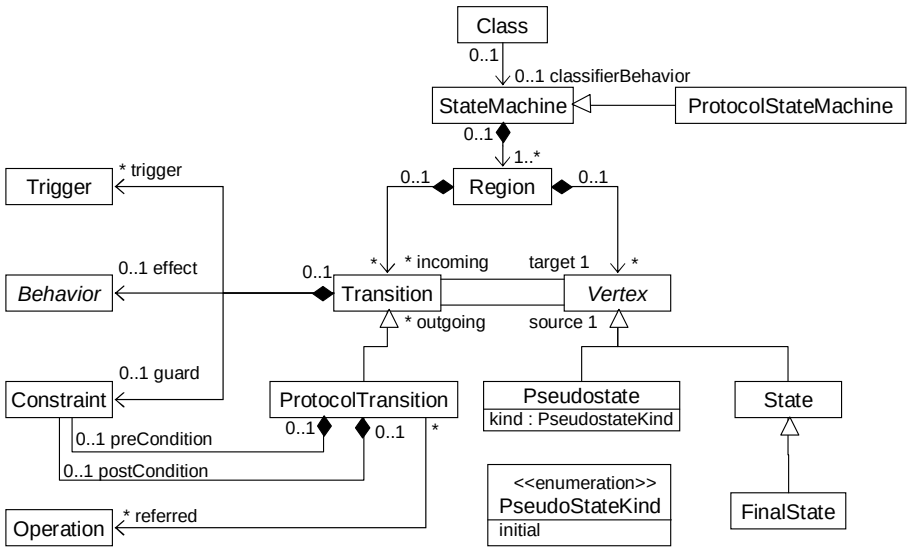


Figure 2.6: Subset of UML State Machines meta-model

A protocol state machine uses a ProtocolTransition, which can be associated with an operation of the Class whose behavior protocol is modeled, a preCondition under which the operation can be called on an object of the Class and a postCondition that must hold for the state to change to the transition target state. According to the UML State Machine specification [UML, 2007a, pg. 523], a protocol state machine "is a convenient way to define a lifecycle for objects".

Figure 2.7 shows an object life cycle for an insurance claim, introduced in Chapter 1, modeled as a protocol state machine. A transition label in a protocol state machine has the following format: [pre-condition] operation / [post-condition]. According to this object life cycle, the operation $Open$ can only be performed on a claim when it is in the initial state, which will always lead to the claim transiting to state $Opened$. When the operation $Evaluate$ is performed on a claim in state $Opened$, state $Granted$ is reached under the condition $claimGranted$ and state $Rejected$ is reached under the condition $claimRejected$. The $claimGranted$ and $claimRejected$ are post-conditions of the two transitions, which means that they are evaluated after the operation $Evaluate$ has been performed. The state $Closed$ must be reached by any claim at the end of its life cycle.

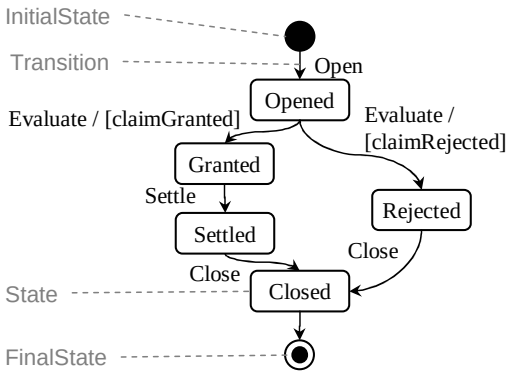


Figure 2.7: Protocol state machine

UML State Machines are suited for detailed modeling of object life cycles during object-oriented software development. Object life cycle modeling in the context of BPM is usually performed on a higher level of abstraction. For example, the object life cycle models defined as part of the IBM Insurance Application Architecture (IAA)[5], a best-practice model collection for the insurance industry, contain transitions labeled with operations only, omitting pre- and post-conditions. In this dissertation, we consider this simplified type of object life cycle models.

This concludes our overview of the background and existing work related to process modeling and object life cycle modeling. In Chapter 3, we formalize the syntax and semantics of the fundamental process and object life cycle modeling elements as the foundation for our framework for integrated process and object life cycle modeling. We next review the state of the art in the modeling using multiple views or *multi-view modeling*.

2.4 Multi-View Modeling

Since the days of SSADM, multi-view or multi-perspective modeling has been promoted in several areas of software engineering. A prominent example is the ViewPoints framework [Finkelstein et al., 1992], which addresses software modeling in a multi-role environment, where each role is responsible for modeling the application under development from a different viewpoint. Each viewpoint is characterized by attributes such as its representation style, its modeling domain, a work plan, a work history record and of course, the model itself. Maintenance of different models with templates describing them as viewpoints facilitates a managed process of distributed modeling throughout software development. The concept of viewpoints has also been applied to the areas of requirements elicitation [Sommerville et al., 1998] and software process modeling [Sommerville et al., 1999].

Multi-view software process modeling is also addressed in [Verlage, 1994], where a set of general requirements for approaches that support multi-view modeling is defined. These requirements include detection of similarity or *overlaps* between views and detection of view *inconsistencies*. Other literature [Grundy et al., 1998, Küster, 2004, Van Der Straeten, 2005, Dijkman, 2006, van Hee et al., 2006] also confirms that view overlaps and inconsistency management are central topics in multi-view modeling. In the methodology proposed in [Küster, 2004] for example, model overlaps are studied during the definition of *consistency* for the models at hand, followed by the spec-

[5] `http://www.ibm.com/industries/financialservices/doc/content/solution/278918103.html`

ification of conditions to check whether consistency holds. On the other hand, [Van Der Straeten, 2005] is concerned with the resolution of inconsistencies detected during consistency checking.

Küster [Küster, 2004] and Van Der Straeten [Van Der Straeten, 2005] both emphasize the role of *model transformations* in multi-view modeling. In the context of MDE, a model transformation refers to the automatic generation of one model from another [Kleppe et al., 2003]. While Küster predominantly suggests using model transformations to map models to a *common semantic domain* for consistency definition, Van Der Straeten uses refactoring transformations to remove inconsistencies from models. In general, model transformations are considered to lie at the heart of MDE approaches [Kleppe et al., 2003, Sendall and Kozaczynski, 2003], where they are used to transform models at the same and different levels of abstraction.

Based on the existing literature, we therefore identify consistency management, inconsistency management and model transformations as the three main topics in multi-view modeling, as illustrated in Figure 2.8(a). These three topics are of close relevance to the solution we propose in this dissertation. In the following, we provide an overview of the state of the art in these areas.

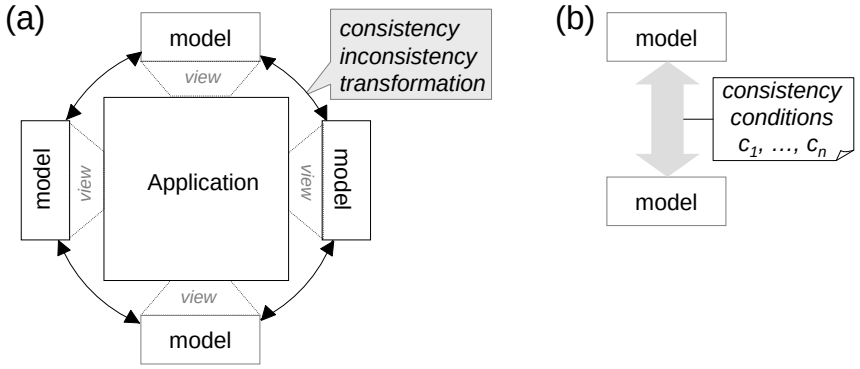


Figure 2.8: (a) Multi-view modeling (b) Consistency conditions

2.4.1 Consistency Management

Several threads of research on model consistency have focused on providing generic frameworks and methods for defining and checking consistency of models. In his PhD dissertation [Küster, 2004], Küster describes a consistency management methodology for object-oriented behavioral models. The importance of model semantics for consistency management is emphasized in this approach, which prescribes mapping the models in question to a common semantic domain as one of the first steps in managing their consistency. *Consistency conditions* are defined to capture the required consistency in terms of semantic and syntactic model elements. Various degrees and types of formalizations are described for different application domains and consistency requirements.

A framework for preserving consistency in multi-view architectural designs for distributed systems is proposed in the PhD dissertation of Dijkman [Dijkman, 2006]. In this framework, different stakeholders are responsible for specifying *consistency rules* for keeping the models that represent their different views consistent. A set of design modeling concepts is predefined in the framework, together with a basic set of consistency rules. Additionally, a language for expressing inter-view consistency rules is provided. In both frameworks, by Küster and by Dijkman, consistency is defined as a set of conditions or

rules, as illustrated in Figure 2.8(b).

Other works in the area have focused on defining consistency with regards to concrete types of models (e.g. [Bhaduri and Venkatesh, 2002, Rasch and Wehrheim, 2003, Van Der Straeten et al., 2003]). As the de facto industry standard for software modeling using multiple views, UML has been the subject of several threads of research on model consistency. The OCL constraints defined in the UML specification have been found insufficient to cover many aspects of consistency between different UML models. Van Der Straeten et al [Van Der Straeten et al., 2003] propose to use description logic for consistency management of Class Diagrams, Sequence Diagrams and State Machines in UML. Several structural and behavioral aspects of consistency are covered, including behavior compatibility of sequence diagrams and state machines. UML models are translated to a reasoning system, which evaluates consistency conditions in description logic on these models. Consistency of UML Activity Diagrams and UML State Machines, which can be used to represent process and object life cycle models respectively, has not been addressed in the existing work.

In the existing literature [Huzar et al., 2005], two main types of consistency are distinguished: *intra-model consistency* and *inter-model consistency*. While intra-model consistency deals with establishing that a plausible relation exists between the elements of one model, inter-model consistency is concerned with relations between several models. In Chapter 4, we address intra-model and inter-model consistency of process and object life cycle models, using the main concepts of the consistency management methodology proposed by Küster [Küster, 2004].

2.4.2 Inconsistency Management

Evaluation of consistency conditions can lead to the detection of inconsistencies. Therefore, the topic of inconsistency management is naturally related to consistency management, but is distinguished in much of the existing literature to emphasize the different set of problems that it addresses. Several existing works in the area of inconsistency management [Nentwich et al., 2003, Van Der Straeten, 2005, Mens and Van Der Straeten, 2006] advocate supporting interactive resolution of model inconsistencies by implementing model transformations that can be invoked by the modeler to automatically change the elements of inconsistent models. Such model transformations are referred to as *resolution actions* [Van Der Straeten, 2005], *repair actions* [Nentwich et al., 2003], *resolution rules* [Van Der Straeten and D'Hondt, 2006] or simply *resolutions* [Mens and Van Der Straeten, 2006]. Figure 2.9(a) illustrates an application of a resolution to two models, as a result of which both models are changed and the set of inconsistencies between them is updated.

One of the problems arising in inconsistency resolution is caused by unwarranted dependencies between inconsistencies and resolutions. These dependencies manifest themselves as *side-effects* of resolution application. Figure 2.9(b) illustrates a situation where applying a resolution to remove inconsistency i_1 also removes other inconsistencies and introduces new ones. Side-effects can also lead to *cycles* in the resolution process, i.e. a situation where the collective side-effects of resolving one inconsistency eventually introduce the same inconsistency.

The problem of resolution side-effects and cycles has only been partially addressed in the existing literature. In [Mens et al., 2006a, Mens et al., 2006b], the AGG graph transformation tool [Taentzer, 2003] is used to detect potential dependencies between different inconsistency resolutions. Inconsistency detection and resolution rules are expressed

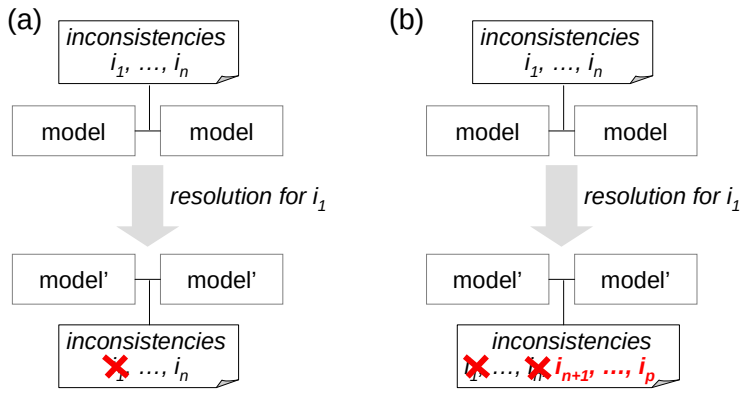


Figure 2.9: Inconsistency resolution

as graph transformation rules in AGG and are then analyzed using critical pair analysis. Analysis results point to potentially conflicting resolutions, resolutions that may induce or expire other types of inconsistencies and potential cycles between resolutions. The work described in [Egyed, 2007] presents a tool-supported approach to fixing inconsistencies in UML models with a special focus on inconsistency resolutions with side-effects. Abstract resolutions (e.g. indicating what model element should be changed, but not how it should be changed) for discovered inconsistencies are automatically determined and displayed to the modeler together with information about possible resolution side-effects.

Other research in the area includes several frameworks for inconsistency management. A conceptual framework for inconsistency management in software development presented in [Nuseibeh et al., 2001] distinguishes between inconsistency detection, diagnosis and handling phases. Classification of inconsistencies is proposed as part of the diagnosis phase, however no concrete classification techniques are given. Another framework is proposed in [Grundy et al., 1998], which presents an approach to managing inconsistencies in multiple-view tools based on Change Propagation and Response Graphs (CPRGs). Software artifacts such as classes and their associated views are stored as components in a graph that captures their interdependencies. Changes to views are either automatically propagated using the CPRG or displayed to the modeler for resolution.

The extensive survey of the existing work and open problems in model inconsistency management described in [Spanoudakis and Zisman, 2001] concludes that one of the most important open research issues is providing more guidance to the modeler for choosing among multiple alternative inconsistency resolutions. The authors argue that resolutions should be ordered based on cost, risk and benefit. They further conclude that existing approaches do not adequately address efficiency and scalability of inconsistency detection in models that change during the resolution process.

Efficiency of consistency checking and inconsistency resolution is one of the aspects addressed by the FUJABA tool suite [Nickel et al., 2000], which supports both manual and automatic incremental inconsistency resolution [Wagner et al., 2003]. Consistency checking rules can be configured by the modeler and organized into different categories in order to support domain- or project-specific consistency requirements. Consistency checking rules and inconsistency resolution rules are specified using graph grammar rules and executed by the FUJABA rule engine. Work on incremental transformations using triple graph grammars [Schürr, 1994] studies the problem of keeping two models synchronized [Giese and Wagner, 2006, Becker et al., 2007]. This is achieved by analyzing changes in one model and applying incremental updates for re-establishing consistency.

In Chapter 5, we address several of the problems of inconsistency resolution mentioned above and demonstrate our solution in the context of process and object life cycle models. We describe how to prioritize inconsistencies for guiding the modeler during the resolution process and present an approach for developing resolutions that facilitates the forecast of resolution side-effects and cycles.

2.4.3 Model Transformations

The concept of a transformation has already been well-established in compiler theory, where high-level programming languages are transformed to an executable machine language. The MDE approaches have adapted the concept of a transformation for the modeling domain [Kleppe et al., 2003]. The input and output of a model transformation are commonly referred to as the *source model* and *target model*, respectively.

As described in several recently published surveys [Czarnecki and Helsen, 2003, Sendall and Kozaczynski, 2003, Czarnecki and Helsen, 2006], the area of model transformations has received a lot of attention from researchers in the past years. The main focus of research has been on the development of generic techniques or languages for the specification of model transformations. Such techniques include those based on graph theory (e.g. VIATRA [Csertán et al., 2002] and GReAT [Agrawal, 2004]), mathematical relations (e.g. QVT [QVT, 2008]) and many others.

Some model transformation languages have been used in case studies to specify mappings between different modeling languages, such as for instance the translation of UML State Machines to Communicating Sequential Processes (CSP) [Heckel et al., 2002] or Simulink Stateflow models to hybrid automata [Agrawal et al., 2004]. Such case studies do not currently include transformations between process and object life cycle models.

Without the use of a specific model transformation technique, Redding et al describe a translation of object-oriented models to process-oriented models [Redding et al., 2007]. Object models based on a custom meta-model of the FlowConnect system are translated to YAWL [van der Aalst and ter Hofstede, 2005]. An object model defines object types, each associated with a state machine and enabled to communicate with other state machines via signals. Signals can spawn new state machines, carry messages between state machines and terminate state machines. The translation is performed in several steps, including the creation of a heuristic net from the object model, translation of the heuristic net to a Petri net, and then the Petri net to YAWL. The translation of the heuristic net to a Petri net is done in the process mining tool ProM [van Dongen et al., 2005]. An additional step of subprocess identification in the generated process model is discussed as future work.

Other work on transformation between process and object life cycle models includes that in the context of the artifact-centric method for developing BPM applications [Nigam and Caswell, 2003, Bhattacharya et al., 2005, Liu et al., 2007]. The authors describe a transformation from the so-called business operations models to Adaptive Business Object (ABO) components [Nandi and Kumaran, 2005, Kumaran et al., 2003]. A business operations model can be considered as a special type of process model that captures how tasks read and write objects stored in data repositories. The control flow is thus implicitly defined via data flow. An ABO component encapsulates behavioral and structural information about an object. The behavior is represented with finite state machines that contain events and actions to allow communication between different ABOs to take place. Structure of the data relevant for the ABO is captured in a data graph, while only references to data locations and not the data themselves are stored in the ABO. The translation of operations models to ABOs is illustrated using an example

in [Bhattacharya et al., 2005], but no detailed description or algorithm is provided.

In Chapter 6, we develop model transformations from process to object life cycle models and vice versa and show that these transformations ensure consistency and minimality of the target models with respect to the source models. By showing that our transformations satisfy these desired properties, we go beyond the related work described above.

With this, we conclude our overview of the related work in the area of multi-view modeling. In the context of BPM, models are predominantly created during the analysis and design phases of the so-called *BPM life cycle* (not to be confused with an object life cycle) and are then used to derive a central part of the implementation of the application being developed. In the next section, we review the phases of the BPM life cycle in more detail and provide an overview of how the transition from design to implementation is supported in the existing approaches.

2.5 From Design to Implementation

The development phases grouping the BPM-related activities are usually depicted as the BPM life cycle, described next.

2.5.1 BPM Life Cycle

Several life cycle models have been proposed in the literature to organize the BPM activities into several phases [Hollingsworth, 2004, zur Muehlen, 2004, Dumas et al., 2005]. Many life cycle models build upon the fundamentals of the software development life cycle [Sommerville, 2006], distinguishing the requirements analysis, design and implementation phases. Continuous improvement of business processes is facilitated by monitoring executing business processes and providing feedback back to the design phase. This is illustrated in Figure 2.10, which shows the BPM life cycle model proposed by zur Muehlen [zur Muehlen, 2004].

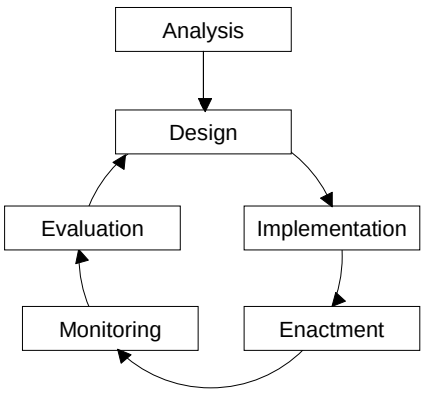


Figure 2.10: BPM life cycle

The *analysis* phase is concerned with analyzing the goals and requirements of the business process to be implemented. The output of this phase is a requirements specification, which comprises a natural language description and/or a high-level process model.

In the *design* phase, all the tasks in the business process and the order in which they need to be performed are identified and captured in a process model. Generally, data required and manipulated by the tasks and the roles responsible for carrying out the tasks are specified in the model. Simulation, validation and verification of the process model

are also usually performed during this phase. The output of the design phase is a detailed process model, which can be used as a basis for the implementation.

The *implementation* phase is concerned with developing an executable implementation of the business process on the basis of the process model designed in the previous phase. The implementation is developed in a language that can be interpreted by a process execution engine. The process model is either used as a blueprint during the implementation phase or the implementation is automatically generated from the process model. Integration with the existing infrastructure is performed. The output of this phase is an executable process implementation.

During *enactment*, the process implementation is deployed on a process engine responsible for the creation and coordination of process instances. *Monitoring* is performed on the executing process instances to compute business-relevant metrics. *Evaluation* of these metrics determines whether the design phase should be entered again to adapt the process model for performance improvement.

In this dissertation, we focus predominantly on the analysis and design phases of the BPM life cycle and cover some of the issues that arise during the transition from the design phase to the implementation phase. In the following, we take a closer look at the different approaches to implementing processes and deriving implementations from models created during the design phase.

2.5.2 Deriving Process Implementations

While the focus of process implementations of the 20th century was on automation of business processes within one organization, in the recent decade it has shifted to distributed and inter-organizational processes. This is facilitated by a new distributed computing paradigm, known as the Service-Oriented Architecture (SOA) [Erl, 2005], which breaks down a business process into reusable functional units called *services*. Therefore, process implementations of today usually capture an orchestration of distributed services. The Business Process Execution Language for Web Services (BPEL4WS or BPEL) [BPEL, 2003] is the standard for defining processes that orchestrate services over standard Internet protocols.

Figure 2.11(a) illustrates that black-box tasks in a process model get associated with their realizations during the implementation phase. In a SOA, tasks are usually loosely-coupled with their realizations via a service layer. A task is associated with a service, which in turn calls various applications and uses various databases.

The transformation of the process layer itself usually requires a transformation from a Platform Independent Modeling (PIM) language to a Platform Specific Modeling (PSM) language. If the two languages are based on similar concepts, the resulting process implementation and the process model may be closely related. An example of such a transformation is a mapping of BPMN process models to BPEL implementations. Both of these languages focus on capturing the flow of control and data between business process tasks and can therefore be referred to as task-centric or activity-centric. However, BPMN is a graph-based language that does not enforce a block-structure on process models, while BPEL is to a large extent a structured language. Therefore, additional structure needs to be introduced during the transformation from languages such as BPMN to BPEL [Lassen and van der Aalst, 2006, Ouyang et al., 2006, Recker and Mendling, 2006]. Nevertheless, the process model and process implementation are still conceptually close in the transformation from BPMN to BPEL.

Figure 2.11(b) illustrates another scenario, where the relationship between the pro-

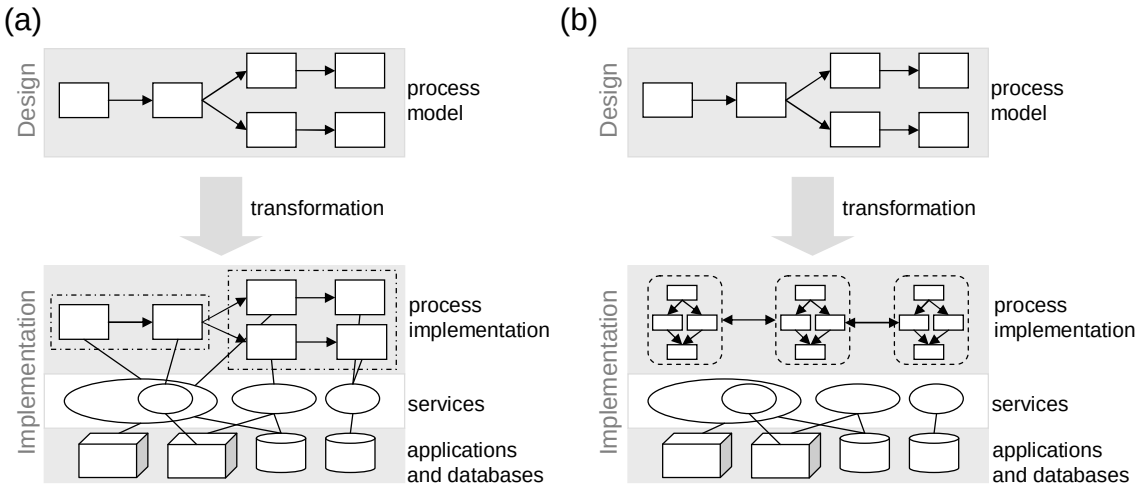


Figure 2.11: Process model and process implementation

cess model and the process implementation is not immediately visible. If a so-called object-centric language (e.g. [van der Aalst et al., 2001, Nandi and Kumaran, 2005]) is used for the implementation, then the process logic is split up into object components during the PIM-to-PSM transformation. Each object component represents a life cycle of one object that participates in the overall process. The components interact for synchronization to ensure that the overall process logic is correctly implemented. Approaches to process implementation that can be described as object-centric include proclets [van der Aalst et al., 2001], ABOs [Nandi and Kumaran, 2005] and data-driven process structures [Müller et al., 2006, Müller et al., 2007].

The transition from design process models to activity-centric process implementations have already received a significant amount of attention [Hauser and Koehler, 2004, Lassen and van der Aalst, 2006, Ouyang et al., 2006, Recker and Mendling, 2006]. However, the derivation of object-centric process implementations from process models has not been studied in detail. An example of deriving ABOs from business operations models is given in [Bhattacharya et al., 2005], but no detailed description of the mapping is provided. In a more recent follow-up work [Kumaran et al., 2008], an algorithm for the derivation of object components from process models is proposed. However, the algorithm simplifies the mapping of control nodes and hence does not ensure full preservation of the behavior captured in the original process model.

In this dissertation, we complement our integration of process and object life cycle modeling for the analysis and design phases by addressing some challenges of deriving object-centric process implementations from process models, as described in Chapter 7.

2.6 Summary and Discussion

In this chapter, we provided the background on the main topics covered in this dissertation, comprising process and object life cycle modeling, model consistency and inconsistency management, model transformations and derivation of process implementations from process models. We have also provided an overview of the most significant related works in these areas.

Given this, we are now ready to present the components of our framework for the integration of process and object life cycle modeling. As discussed in Chapter 1, these

components comprise the following:

- Syntax and semantics of process and object life cycle models;
- Consistency, focusing on defining and evaluating consistency of process and object life cycle models;
- Inconsistency resolution, concerned with the resolution of inconsistencies between process and object life cycle models;
- Model transformations, comprising object life cycle extraction and process model generation;
- Transition from design to implementation, focusing on the derivation of object-centric process implementations.

Each of the following five chapters is dedicated to describing one of the above-listed components.

Chapter 3

Syntax and Semantics

Establishing the relationship between different model types and defining their consistency requires a clear understanding of the syntax and semantics of the underlying modeling languages. Therefore, this chapter is dedicated to explaining and precisely defining the syntax and semantics of process and object life cycle models. We begin by giving a brief overview of how syntax and semantics of a language are commonly specified in Section 3.1. With respect to the syntax and semantics of process models, we specifically focus on aspects of data flow and object state modeling, which are essential for establishing the relationship with object life cycle models and at the same time are not adequately addressed in the existing literature. We formalize these aspects by extending an existing definition of a generic process modeling representation, called a workflow graph, and its control-flow semantics in Section 3.2. For object life cycle models, we choose a syntactic and semantic definition based on automata theory, suitable for using these models as protocols of object state evolution in Section 3.3.

3.1 Syntax and Semantics of a Language

Syntax, semantics and pragmatics are the three fundamental aspects that describe a language [Slonneger and Kurtz, 1995], be it a natural language, a programming language or a graphical modeling language. *Syntax* defines the relations between the different elements of a language, providing a structural description of legal constructs in a language without a consideration of what the constructs mean. On the other hand, *semantics* establishes the meaning of different language constructs that are legal with respect to the syntax. Finally, *pragmatics* deals with the aspects of a language that involve its users, such as for example ease and efficiency of use. In this dissertation, we are only concerned with the syntax and semantics of process and object life cycle models, leaving pragmatics out of our scope.

The syntax of a language can be described by a *grammar* [Hopcroft et al., 2006, Slonneger and Kurtz, 1995], which comprises rules for assembling valid constructs from the elementary language elements. The Backus-Naur Form (BNF) [Backus et al., 1963] represents a special notation for capturing grammars for programming languages. Grammars, more specifically graph grammars [Ehrig et al., 1999], have also been applied to describing the syntax of modeling languages (e.g [Karsai et al., 2003, Hermann et al., 2008]). However, meta-modeling (cf. Chapter 2) has become a much more popular approach to capturing the syntax of a modeling language in a graphical way.

There is a distinction between the concrete syntax and the abstract syntax of a language [Slonneger and Kurtz, 1995]. *Concrete syntax* refers to the physical representation of the language exposed to its users, such as symbols of a programming language or graphical symbols of a modeling language. On the other hand, *abstract syntax* is an internal representation of a language used for automated language manipulation. While BNF is used to describe a language using its concrete syntax, a meta-model represents the structure of a modeling language independently from its graphical notation using its abstract syntax.

The execution semantics of a programming language or a behavioral model can be specified using one of the following types of semantics: operational, denotational or axiomatic semantics [Nielson and Nielson, 1992]. *Operational semantics* describes the meaning of a construct as a sequence of computation steps, making it explicit how the effect of the construct's execution is computed. *Denotational semantics* represents the effect of executing a particular construct in terms of mathematical objects, abstracting from some details of the computation. Finally, *axiomatic semantics* only focuses on assertions that hold after the execution of a construct, ignoring some details of the computation and the overall effect of the construct's execution.

In this dissertation, we use the graphical notations of UML Activity Diagrams and UML State Machines as the concrete syntax for process and object life cycle models, respectively. Although we presented extracts of the UML meta-model relevant for process and object life cycle modeling in Chapter 2, in this chapter we turn to a more concise set-based representation of the abstract syntax for these models. We define the execution semantics of process models in the form of a *transition semantics* [Plotkin, 1981], which is a special type of operational semantics. For object life cycle models, we do not define an execution semantics, but rather capture the semantics of these models as a set of conditions that can be evaluated with respect to the execution of another system.

3.2 Syntax and Semantics of a Process Model

Our aim in this dissertation is to establish the relationship between process and object life cycle models, to provide an integrated modeling method for these two model types and to demonstrate its value. To obtain generally applicable results, we do not intend to focus on one specific process modeling language, but rather address the most fundamental aspects of process models.

The fundamental process modeling aspects first of all include modeling constructs that are already common to most process modeling languages. As confirmed by the existing evaluations of process modeling languages performed on the basis of the so-called workflow patterns (e.g [Dumas and ter Hofstede, 2001, Wohed et al., 2005, Wohed et al., 2006])[1], there is a core set of control-flow constructs that are incorporated into all of the existing process modeling languages. These are called basic control-flow patterns [van der Aalst et al., 2003, Russell et al., 2006a] and comprise sequence (WP1), parallel split (WP2), synchronization (WP3), exclusive choice (WP4) and simple merge (WP5). In UML Activity Diagrams for example, a sequence of nodes is simply modeled by connecting them with edges, a parallel split is modeled as a fork node, a synchronization as a join node, an exclusive choice as a decision node, and a simple merge as a merge node (cf. Section 2.2 of Chapter 2).

[1] A complete overview of the pattern-based evaluation results for several process modeling languages is provided online: `http://www.workflowpatterns.com`.

Evaluations of the existing process modeling languages based on workflow patterns also show that various aspects of data-flow modeling are not yet as widely supported (e.g. [Russell et al., 2006b, Wohed et al., 2006]). However, since all actions in a process generally require data to execute, we also consider data-flow aspects to be fundamental in process models. Among all the possible data-flow constructs [Russell et al., 2004, Russell et al., 2005], we focus on a set of basic constructs for modeling the manipulation of data by single actions and the transfer of data between the nodes in a process model. In addition, since the specification of object states in a process model is key to establishing the relationship between process and object life cycle models and is already supported in several existing process modeling languages, we also consider it to be a fundamental aspect of process models.

Figure 3.1 shows a claims handling process model in the UML Activity Diagram notation, containing control flow, data flow and a state specification for the $Claim$ object type. Control flow is captured by the edges in the model, data flow is specified by the pins and the edges together, while object states are indicated in square brackets connected to the pins with dotted lines.

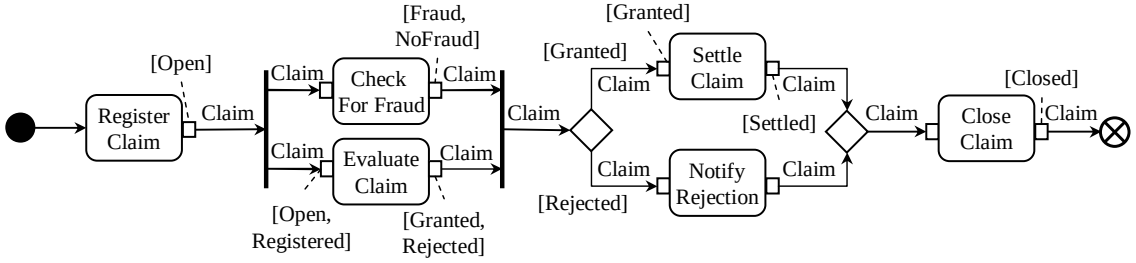


Figure 3.1: Process model with data flow and object states

The concrete syntax as such is not essential in our work, since BPMN or another notation can be used to visually represent process models such as the one shown in Figure 3.1. The abstract syntax and the semantics are of major importance however, since they respectively define the structure of valid models and how models are to be interpreted.

The semantics of the fundamental control-flow constructs (WP1-WP5) is already well-understood and unambiguously described in the specifications of the existing process modeling languages. For example, according to the semantic specification of UML Activity Diagrams, the claims handling process model in Figure 3.1 executes as follows: $Register\ Claim$ is the first action to be executed, after which the $Check\ For\ Fraud$ and $Evaluate\ Claim$ actions are carried out in parallel; this is followed by the execution of either the $Settle\ Claim$ action or the $Notify\ Rejection$ action; and finally the $Close\ Claim$ action is performed to complete the execution of the process. It is, however, not as clear as to what happens to the $Claim$ object and its state during the execution of the process. The questions left unanswered include the following:

- Does an object of type $Claim$ exist before the process execution begins or is it created inside this process?
- Is the $Claim$ object received on the input pin of the $Check\ For\ Fraud$ action the same object that is produced on the output pin of this action?
- What state transitions does the $Evaluate\ Claim$ action induce during the execution of this process? For example, does it induce the transition from $Registered$ to $Granted$?

- Do the $Check\ For\ Fraud$ and $Evaluate\ Claim$ actions change the state of the same $Claim$ object or different $Claim$ copies?

- $Notify\ Rejection$ has input and output pins of type $Claim$, but no states specified on the pins. Does this mean that it does not induce any state changes for the $Claim$ or that it can induce any arbitrary state changes?

Even more of such questions would arise if we had used the BPMN or EPCs notations, since the data-flow semantics of these languages is even less complete than that of UML Activity Diagrams. These questions primarily relate to two aspects of a data-flow semantics that we refer to as object manipulation and object passing. The *object manipulation* semantics is concerned with the use of objects by single actions, including object creation, reading, update and deletion. The *object passing* semantics relates to how objects are transferred between actions and other nodes in a process model, detailing whether objects are passed by reference or by value. In addition to an imprecise specification of object manipulation and passing, an unclear semantics of object state specifications in process models is another reason for the ambiguity that leads to the above-mentioned questions.

In this chapter, we define a data-flow and object state specification semantics that provides precise answers for questions such as the ones mentioned above. As a foundation, we use an existing definition of a generic process modeling representation, called a workflow graph, and its control-flow semantics. For the concrete syntax, we continue using the UML Activity Diagrams with a custom notation for the splits of object flows connected to repositories (see Section 2.2.2 of Chapter 2). The resulting configuration of the syntax and semantics of process models used in this dissertation is illustrated in Figure 3.2, where our contributions are indicated with the two shaded boxes.

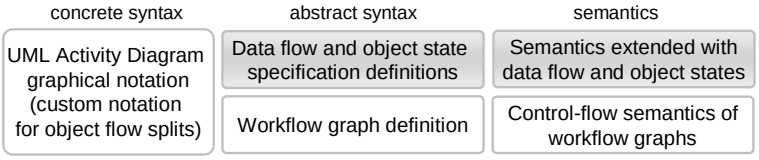


Figure 3.2: Process model syntax and semantics used in this dissertation

We begin by introducing workflow graphs and their control-flow semantics in the following section.

3.2.1 Existing Control-Flow Syntax and Semantics

A *workflow graph* [Sadiq and Orlowska, 2000] is a language-independent representation of a process model that covers the fundamental control-flow process modeling constructs (WP1-WP5). A subset of process models represented in languages such as UML Activity Diagrams can be mapped to workflow graphs, see for example [Favre, 2008]. Subprocess hierarchy, employed in many process modeling languages, is flattened during such mappings to workflow graphs.

We use a definition of a workflow graph that is based on the one published in the work of Vanhatalo et al [Vanhatalo et al., 2007]. Instead of the term "activity", we use the term "action" to avoid confusion with the UML Activity Diagram concept of an activity. Additionally, we augment the original definition by introducing two functions in and out for a convenient means of referring to the incoming and outgoing edges of a node.

Definition 1 (Workflow graph)**.** *A* workflow graph *is a directed graph $G = (N, E)$, where a node $n \in N$ is one of the following: a* start node*, a* stop node*, an* action*, a* fork*, a* join*, a* decision *or a* merge*. The functions $in, out : N \to \mathcal{P}(E)$ map a node to the set of its* incoming *and* outgoing edges*, respectively. The following conditions hold with respect to a workflow graph $G = (N, E)$:*

- *there is exactly one start node and exactly one stop node in N;*
- *the start node has no incoming edges and exactly one outgoing edge, whereas the stop node has exactly one incoming edge but no outgoing edges;*
- *each fork and each decision has exactly one incoming edge and two or more outgoing edges, whereas each join and each merge has exactly one outgoing edge and two or more incoming edges;*
- *each action has exactly one incoming and exactly one outgoing edge;*
- *each node $n \in N$ is on a path from the start node to the stop node.*

The claims handling process model shown in Figure 3.1 can be mapped to a workflow graph by simply removing all the elements related to data flow, as shown in Figure 3.3, where action names are abbreviated for conciseness.

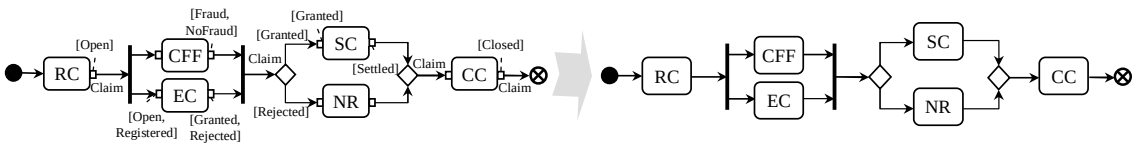


Figure 3.3: Obtaining a workflow graph

The execution semantics of a workflow graph [Vanhatalo et al., 2007] is defined in terms of a flow of *tokens*. An *execution state* of a workflow graph (not to be confused with a state of an object) is represented by a distribution of tokens on the edges of the graph[2].

Definition 2 (Execution state)**.** *Given a workflow graph $G = (N, E)$, its* execution state *is a mapping $w : E \to \mathbb{N}$ that assigns natural numbers to all edges in G. The number assigned to an edge in an execution state w represents the number of tokens carried by that edge in w.*

The semantics of the various nodes is defined as follows. An action, a fork, and a join remove one token from each of its incoming edges and add one token to each of its outgoing edges. A decision removes a token from its incoming edge and adds one token to one of its outgoing edges. Since workflow graphs abstract from decision logic, the choice of the outgoing edge is performed nondeterministically. A merge nondeterministically chooses one of its incoming edges on which there is at least one token, removes one token from that edge, and adds a token to its outgoing edge. In the following, this semantics is defined formally (paraphrased from [Vanhatalo et al., 2007]).

[2]We use the term "execution state" instead of the term "state" originally used in [Vanhatalo et al., 2007] to avoid confusion with the notion of an object state. Also for reasons of disambiguation, we call the token mapping w instead of s in the definition of the execution state.

Definition 3 (Control-flow semantics)**.** *Let w and w' be two execution states of a workflow graph $G = (N, E)$ and $n \in N$ be a node that is neither a start nor a stop node. The change of the execution state from w to w' by the execution of node n is written as $w \xrightarrow{n} w'$. The following three rules define $w \xrightarrow{n} w'$ for different types of nodes:*

1. *n is an action, fork or join and*
$$w'(e) = \begin{cases} w(e) - 1 & e \in in(n), \\ w(e) + 1 & e \in out(n), \\ w(e) & \textit{otherwise.} \end{cases}$$

2. *n is a decision and there exists an outgoing edge $e' \in out(n)$ of n such that*
$$w'(e) = \begin{cases} w(e) - 1 & e \in in(n), \\ w(e) + 1 & e = e', \\ w(e) & \textit{otherwise.} \end{cases}$$

3. *n is a merge and there exists an incoming edge $e' \in in(n)$ of n such that*
$$w'(e) = \begin{cases} w(e) - 1 & e = e', \\ w(e) + 1 & e \in out(n), \\ w(e) & \textit{otherwise.} \end{cases}$$

The initial and terminal execution states of a workflow graph are defined as follows.

Definition 4 (Initial and terminal execution states)**.** *Given a workflow graph G, its* initial execution state *is the execution state w_i that has exactly one token on the outgoing edge of the start node and no tokens elsewhere. The* terminal execution state *of G is the state w_t that has exactly one token on the incoming edge of the stop node and no tokens elsewhere.*

Furthermore, several additional concepts related to the execution of a workflow graph are defined.

Definition 5 (Activated node, execution sequence)**.** *A node n is said to be* activated *in an execution state w if there exists another execution state w' such that $w \xrightarrow{n} w'$. A sequence of node executions $w_0 \xrightarrow{n_1} w_1 \dots w_{k-1} \xrightarrow{n_k} w_k$ is called an* execution sequence*.*

Definition 6 (Reachable execution state)**.** *An execution state w' is* reachable *from an execution state w, denoted $w \xrightarrow{*} w'$, if there exists a (possibly empty) finite execution sequence $w_0 \xrightarrow{n_1} w_1 \dots w_{k-1} \xrightarrow{n_k} w_k$ such that $w_0 = w$ and $w_k = w'$.*

Figure 3.4 illustrates the application of the three execution rules in Definition 3 using the claims handling process model as an example. Each workflow graph in the figure represents an execution state, where tokens are shown as gray circles marked with numbers and activated nodes are marked with a tick.

In Figure 3.4, starting from the initial execution state w_1, rule 1 is applied five times to first execute action RC, followed by the fork, then action CFF, action EC and finally the join. Actions on the two parallel paths are interleaved, so in another execution sequence, action EC could be executed before CFF. After the execution of the join, the decision becomes activated in w_6 and rule 2 is applied to execute the decision. In this example, the decision places the token on the upper outgoing edge, therefore activating action SC in w_7. After that, rules 1, 3 and 1 are applied, after which the terminal execution state w_{10} is reached.

In the following, we first introduce the definitions of data flow and object states, and then extend the control-flow execution semantics of workflow graphs.

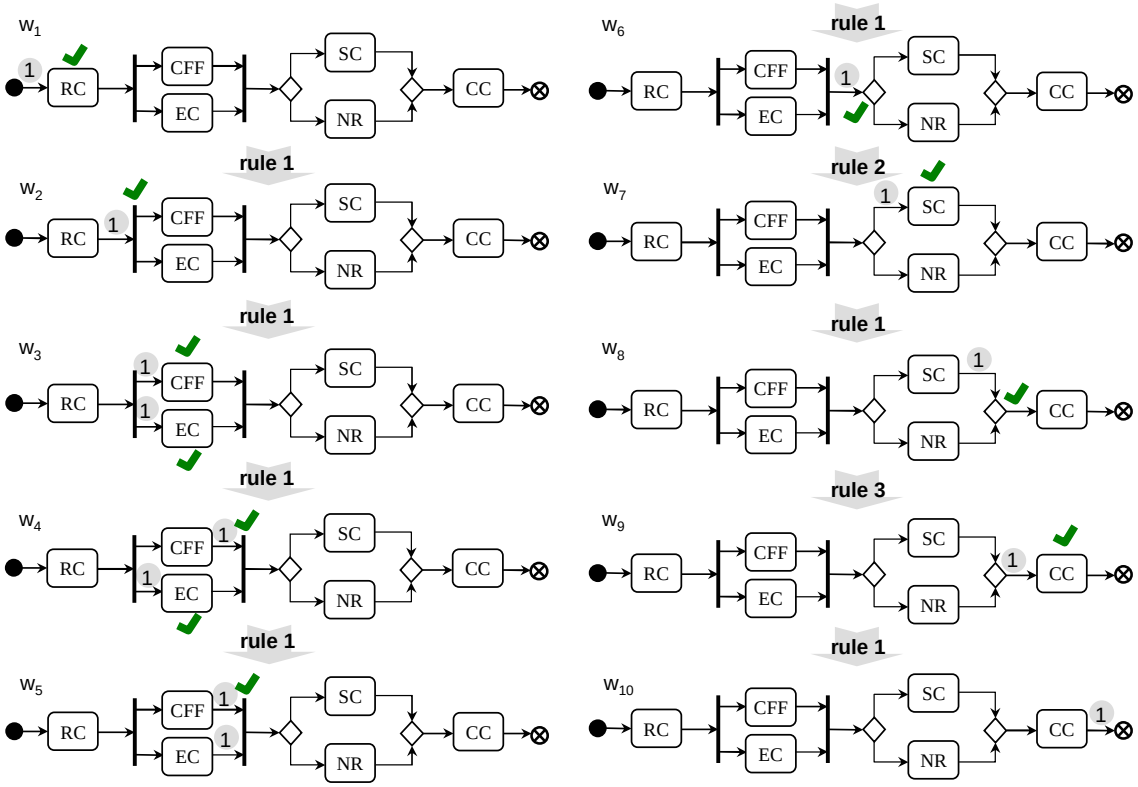


Figure 3.4: Example of workflow graph execution

3.2.2 Extending Abstract Syntax with Data Flow and Object States

Many process modeling languages allow one to model data flow either explicitly by having *typed edges* (such as the example in Figure 3.1), or implicitly by only modeling input and output data requirements of actions [Russell et al., 2005, Sadiq et al., 2004]. Explicit data flow implies that data is routed from one action to another, and therefore we refer to it as *routed data flow*. Implicit data flow assumes that there is a central data repository from which actions read data and to which they write data, and hence we call it *repository data flow*. Since the presence of both types of data-flow modeling is wide-spread, in particular they are supported in both UML Activity Diagrams and BPMN, we provide a formalization of both. Additionally, a mixture of repository and routed data flow within one process model is supported in some modeling languages. For clarity, we assume that a given process model uses only one type of data-flow modeling. Hence, we provide two separate definitions for the data flow of a workflow graph.

Repository Data Flow

We define a data flow for a given workflow graph using a set of object types and a set of functions that relate elements of the workflow graph with these object types. In this way, adding a data flow to a workflow graph leaves the original definition of the workflow graph intact.

In our definition of the repository data flow, each repository is represented by an object type. A node n reading from a repository of type t is represented by n having a data input of type t. Analogously, a node n updating a repository of type t is represented by n having a data output of type t. This is formalized in the following definition.

Definition 7 (Repository data flow)**.** *Given a workflow graph $G = (N, E)$ and a set of object types T,* repository data flow *for G is defined using two functions $datain, dataout : N \rightarrow \mathcal{P}(T)$ that map a node to sets of object types that represent its data inputs and outputs, respectively. Given a node $n \in N$, the following conditions hold:*

- *$datain(n)$ and $dataout(n)$ are empty if n is not an action or a decision;*
- *if n is a decision, $|datain(n)| \leq 1$ and $datain(n) = dataout(n)$.*

In repository data flow, data inputs and outputs of an action are graphically represented using the notation introduced in Section 2.2.2 of Chapter 2, as shown in Figure 3.5. In this example, $datain(a) = \{t_{11}, ..., t_{1p}\}$ and $dataout(a) = \{t_{21}, ..., t_{2q}\}$. According to Definition 7, decisions can also have data inputs and outputs. These data inputs and outputs do not have an explicit graphical representation, but are later used to define control-flow routing based on object states. As we later only consider simple decisions that route control-flow based on the state of objects of one type, the data inputs and outputs of a decision can comprise at most one object type.

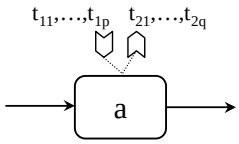


Figure 3.5: Notation for repository data flow

In UML Activity Diagrams, a process model can have several repositories of the same object type, in which case repositories are uniquely identified by their name and type. In a mapping of such process models to our repository data flow, each repository with a unique name and type would be mapped to a different object type.

Routed Data Flow

Routed data flow is defined as an assignment of object types to edges of a given workflow graph. In this type of data flow, an object type no longer represents a repository, but defines the type of objects that can traverse a given edge. Data inputs and outputs of a node are defined on the basis of its incoming and outgoing edges.

Definition 8 (Routed data flow)**.** *Given a workflow graph $G = (N, E)$ and a set of object types T, a* routed data flow *for G is defined using a partial function $type : E \rightarrow T$ that maps an edge to an object type. Functions $datain, dataout : N \rightarrow \mathcal{P}(T)$ are defined to map a node $n \in N$ to its data inputs and outputs based on the types of incoming and outgoing edges of n. Given a node $n \in N$, the following conditions hold:*

- *given an edge $e \in out(n)$, $type(e)$ is undefined if n is the start node;*
- *all incoming and outgoing edges of n are mapped to the same type or no type if n is a decision, a merge, a fork or a join;*
- *data inputs and outputs of n comprise the types of the incoming and outgoing edges of n, respectively:*
  - *$\forall t \in T.\ (\exists e \in in(n).\ type(e) = t) \Leftrightarrow t \in datain(n)$;*
  - *$\forall t \in T.\ (\exists e \in out(n).\ type(e) = t) \Leftrightarrow t \in dataout(n)$.*

Typed edges are graphically represented as UML Activity Diagram object flows connected to actions via pins, as shown in Figure 3.6(a) (cf. Section 2.2.2 of Chapter 2). In this example, $datain(a) = \{t_1\}$ and $dataout(a) = \{t_2\}$. Typed edges connected to nodes other than actions are represented as usual, see for example Figure 3.6(b). In this example, $datain(d) = dataout(d) = \{t\}$.

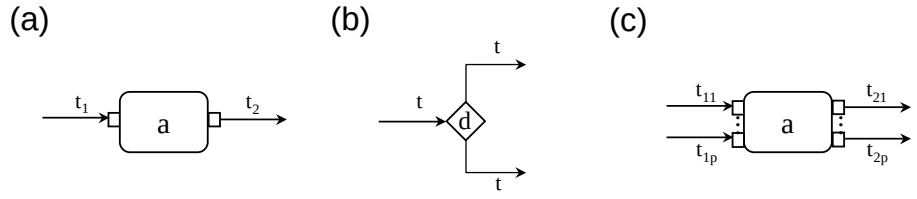


Figure 3.6: Notation for routed data flow

Since workflow graphs only allow one incoming and one outgoing edge per action, routed data flow of a workflow graph can only define one data input and one data output for an action. This is too restrictive, since it is common for actions in a process model to manipulate several object types. Therefore, we introduce a relaxed workflow graph that allows multiple incoming and outgoing edges for actions, so that multiple data inputs and outputs can be represented as illustrated in Figure 3.6(c).

Definition 9 (Relaxed workflow graph with routed data flow)**.** *A* relaxed workflow graph *is a directed graph $G = (N, E)$ that fulfils all properties of a workflow graph, except that its actions can have multiple incoming and outgoing edges. Given routed data flow for G using a set of object types T, each incoming/outgoing edge of an action has either no type or a type that is unique within the set of all typed incoming/outgoing edges of that action.*

If the types of edges are ignored, a relaxed workflow graph can be executed according to the semantics given in Definition 3. In such a setting, actions with multiple incoming or outgoing edges can be regarded as implicit joins or forks, respectively. In the following, we primarily focus on workflow graphs as process models, and generalize the results to relaxed workflow graphs where necessary.

Object State Specification

States are the only attributes of objects that we include in our formalization, abstracting from the values of other attributes that objects may be associated with. As defined in the following, each object type is associated with a set of states that are valid for objects of this type.

Definition 10 (Object type states)**.** *Let a set of object types T be given. Given an object type $t \in T$, we denote its* object type states *as S_t. Sets of object type states $S_{t_1}, ..., S_{t_n}$, where $t_i \in T$ for $1 \leq i \leq n$, form a partitioning of their superset S_T.*

In UML Activity Diagrams and BPMN, object states can be attached to data inputs and outputs of nodes or to typed edges, depending on what type of data-flow modeling is assumed. In our formalization of repository and routed data flow, we can refer to data inputs and outputs of a node using the $datain$ and $dataout$ functions in both data-flow types. Therefore, we associate states with data inputs and outputs of a node in our formalization, referring to them as *accepted* and *produced states*, respectively.

Definition 11 (Accepted and produced states)**.** *Let a workflow graph $G = (N, E)$ with repository or routed data flow using a set of object types T be given. We define functions $acpt : N \times T \rightarrow \mathcal{P}(S_T)$ and $prod : N \times T \rightarrow \mathcal{P}(S_T)$. The function $acpt$ maps a node $n \in N$ and an object type $t \in T$ to a set of* accepted states*, in which n can accept received objects of type t. The function $prod$ maps a node $n \in N$ and an object type $t \in T$ to a set of* produced states*, in which n can produce objects of type t. Given a node n and an object type t, the following conditions hold:*

- *$acpt(n, t) \subseteq S_t$ and $prod(n, t) \subseteq S_t$;*
- *$acpt(n, t)$ is defined if and only if $t \in datain(n)$;*
- *$prod(n, t)$ is defined if and only if $t \in dataout(n)$.*

Accepted and produced states are indicated in square brackets next to object types that label data inputs and outputs (see Figure 3.7(a)) or edges (see Figure 3.7(b)). In both examples, $acpt(a, t) = \{s_{11}, ..., s_{1p}\}$ and $prod(a, t) = \{s_{21}, ..., s_{2q}\}$.

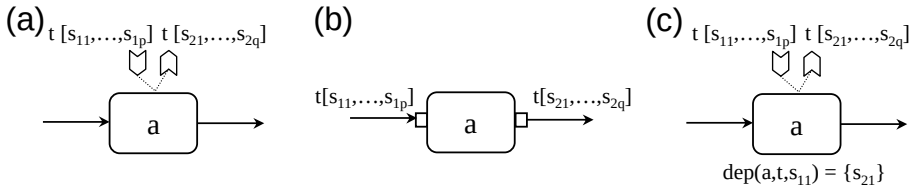


Figure 3.7: Notation for accepted and produced states, and dependency state sets

We also provide a means of capturing dependencies of an output state of an object produced by a particular action on the input state of that object accepted by the action. This aspect is currently not explicitly supported in UML Activity Diagrams or BPMN, but is necessary for precise process modeling with object states to capture situations where a state accepted by an action may only lead to some of the produced states. We introduce the notion of a *dependency state set* to represent this.

Definition 12 (Dependency state set)**.** *Let a workflow graph $G = (N, E)$ with repository or routed data flow using a set of object types T be given. We define a function $dep : N \times T \times S_T \rightarrow \mathcal{P}(S_T)$ to map a node $n \in N$, an object type $t \in T$ and a state $s \in S_T$ to a set of states called a* dependency state set *that comprises produced states of t by n that can result from n accepting an object of type t in state s. Given a node n, an object type t and a state $s \in S_T$, the following conditions hold:*

- *$dep(n, t, s)$ is defined if and only if $s \in acpt(n, t)$;*
- *$\forall s' \in dep(n, t, s).\ s' \in prod(n, t)$.*

Given a node n, an object type t and a state s, we say that $dep(n, t, s)$ is trivial if $dep(n, t, s) = prod(n, t)$. Non-trivial dependency state sets are indicated below the corresponding action, as shown in Figure 3.7(c). In this example, the state s_{11} has a dependency state set $\{s_{21}\}$, which means that if a receives an object of type t in state s_{11}, the state of the produced object by a of type t must be s_{21}.

Apart from the specification of accepted and produced states for actions, object states can also be used to model data-based decision evaluation. For this, we introduce the notion of an *edge condition* that makes it possible to associate object states with outgoing edges of decisions.

Definition 13 (Edge condition)**.** *Let a workflow graph $G = (N, E)$ with repository or routed data flow using a set of object types T be given. We define a function $cond : E \times T \rightarrow \mathcal{P}(S_T)$ to map an edge $e \in E$ and an object type $t \in T$ to a set of states called an* edge condition*, in which e requires objects of type t to be. Given a node n, an edge $e \in out(n)$ and an object type t, the following conditions hold:*

- $cond(e, t) \subseteq S_t$*;*
- $cond(e, t)$ *is defined if and only if n is a decision;*
- $cond(e, t)$ *is defined if and only if $t \in dataout(n)$.*

Edge conditions are graphically represented as an annotation of an edge in the form of an object type with states next to it in square brackets. Figures 3.8(a) and (b) respectively show examples of edge conditions in repository and routed data flow, where $cond(e_1, t) = \{s_{11}, ..., s_{1p}\}$ and $cond(e_2, t) = \{s_{21}, ..., s_{2q}\}$. Since in routed data flow, an edge condition for a given edge is only defined for the type of the edge, type labels can be omitted from the graphical representation, as shown in Figure 3.8(c).

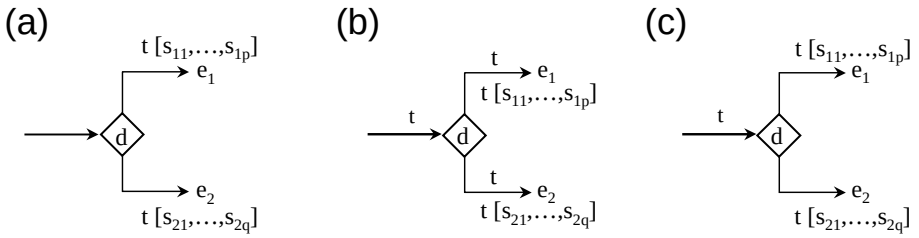


Figure 3.8: Notation for edge conditions

A *state specification* for a given process model is then defined as a specification of accepted and produced states, dependency state sets and edge conditions for the nodes in the process model.

Definition 14 (State specification)**.** *Given a workflow graph $G = (N, E)$ with repository or routed data flow using a set of object types T, a* state specification *for G is defined using functions $acpt$, $prod$, dep and $cond$ defined in Definitions 11-13.*

For convenience, a state specification may be specified only partially in a process model, in which case a default assignment is applied.

Definition 15 (Default state specification assignment)**.** *Let a workflow graph $G = (N, E)$ with repository or routed data flow using a set of object types T and a partial state specification for G be given. The* default state specification *is assigned for each node $n \in N$ and for each object type $t \in T$ as follows:*

1. *first, default values are assigned to unspecified accepted states and edge conditions:*
   - $acpt(n, t) = S_t$ *if $acpt(n, t)$ is unspecified;*
   - $cond(n, t) = S_t$ *if $cond(n, t)$ is unspecified.*
2. *then, default values are assigned to unspecified produced states:*
   - $prod(n, t) = acpt(n, t)$ *if n is an action and $prod(n, t)$ is unspecified;*
   - $prod(n, t) = \bigcup_{e \in out(n)} cond(e, t)$ *if n is a decision and $prod(n, t)$ is unspecified;*

3. *finally, default values are assigned to unspecified dependency state sets for n, t and each state $s \in acpt(n,t)$:*
   - $dep(n,t,s) = \{s\}$ *for all* $s \in acpt(n,t)$ *if* $prod(n,t)$ *was unspecified in the original state specification;*
   - $dep(n,t,s) = prod(n,t)$ *otherwise if* $dep(n,t,s)$ *is unspecified.*

For example, in Figure 3.7(a), $dep(a,t,s_{1p}) = \{s_{21},...,s_{2q}\}$ is obtained as a result of the default state specification assignment according to Definition 15.

In this section we have defined the syntax of data flow and object states, using the workflow graph definition as a foundation. We next proceed to establishing the semantics of these constructs.

3.2.3 Informal Description of Semantics for Data Flow and Object States

We now describe how repository and routed data flow and an object state specification affect the execution semantics of a process model. Most importantly, we extend the notion of an execution state to include a set of objects that exist at a particular point in time during the execution of a process model. In a particular execution state, each object is associated with a type and a state. In the following, we first describe the semantics of object manipulation and object passing informally and then extend the control-flow semantics to capture it.

Object Manipulation

In object-oriented data modeling [Kilov, 1990], four main object manipulation operations are commonly identified: create, read, update and delete, which are collectively referred to as CRUD. These operations also correspond to the main data manipulation functions implemented in relational databases [Date, 2000]. In the following, we show how three of these four object manipulation operations are represented by different assignments of data inputs and outputs to actions in a process model. We do not explicitly represent deletion of objects, assuming that this is a dedicated task performed outside of the usual business process logic captured in process models.

Given an action a and an object type t, four assignments of data inputs and outputs of type t to a are possible using repository or routed data flow defined in the previous section, as shown in Figure 3.9(a)-(d). Since the four cases can be represented in both types of data flow, we avoid the concrete graphical notations of data inputs and outputs for repository and routed data flow, and use an abstract one instead (thick gray arrows).

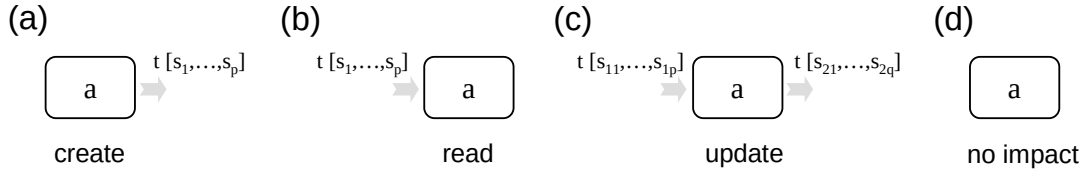


Figure 3.9: Object manipulation

- *create:* A creation of a new object of type t by an action a is represented by assigning action a with a data output of type t and no data inputs of type t, as illustrated in Figure 3.9(a). After the execution of action a, a new object of type t in one of its produced states $s_1, ..., s_p$ is created.

- *read:* An action a that reads an object of type t without updating the object is represented by assigning action a with a data input of type t and no data outputs of type t, as shown in Figure 3.9(b). Action a needs to receive an object of type t in one of its accepted states $s_1, ..., s_p$ to begin execution.

- *update:* An update of an object of type t by an action a is represented by assigning action a with both, a data input and a data output of type t, as illustrated in Figure 3.9(c). In this case, action a needs to receive an object of type t in one of its accepted states $s_{11}, ..., s_{1p}$ to execute, and after the execution of a, the state of the received object is updated to one of the produced states $s_{21}, ..., s_{2q}$.

  Generally, an object update corresponds to a change of the object's state. However, there are some special cases shown in Figure 3.10(a) and (b) that do not lead to an object state change. Given an action a and an object type t, a does not change the state of objects of type t if $acpt(a,t) = prod(a,t) = \{s_1, ..., s_p\}$ and $dep(a,t,s_i) = \{s_i\}$ for $1 \leq i \leq p$ (see Figure 3.10(a)). The example shown in (b) is a special case of (a), where there is only one state in the accepted and produced states of action a for object type t.

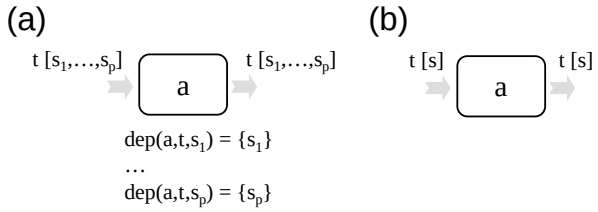


Figure 3.10: Special cases of update with no state change

- *no impact:* An action a that does not manipulate objects of type t is naturally not assigned any data inputs or outputs of this type.

The described object manipulation semantics is comparable to the interpretation of input and output object flows in UML Activity Diagrams outlined by Eshuis in his PhD dissertation [Eshuis, 2002, pg. 110]. Eshuis also considers the deletion of objects to be outside the business process logic.

In our object manipulation semantics, accepted states place additional constraints on when an action can execute, while produced states specify constraints that have to hold after an action has executed. Therefore, accepted and produced states of an action can be seen as a special type of pre-conditions and post-conditions of an action, respectively.

In repository and routed data flow, data inputs and outputs can also be assigned to decisions. This allows us to assign edge conditions to outgoing edges of decisions. The execution semantics of a decision is affected such that all edge conditions for at least one of the outgoing edges of a decision need to hold for the decision to execute, and only such an edge can receive a token as a result of a decision execution. In this way, edge conditions facilitate decision evaluation based on object states, which is a special case of the so-called data-based routing from data workflow patterns [Russell et al., 2005].

For example, considering Figure 3.8(a), the decision d can execute if either all objects of type t_1 are in one of the states $s_{11}, ..., s_{1p}$ or all objects of type t_2 are in one of the states $s_{21}, ..., s_{2p}$. Whether one or several objects of the same type can exist in an execution state of a process model is defined by our interpretation of the object passing semantics, described next.

Object Passing

Similar to the way many programming languages offer two ways of passing arguments to a function (e.g. [Mitchell and Apt, 2001, Pierce, 2002]), objects can be passed either *by reference* or *by value* between the nodes of a process model. Passing an argument by value (also sometimes called pass-by-copy) involves first making a copy of the variable and passing that copy to a function, so that if the copy is changed inside the function, the original variable is not affected. On the other hand, passing an argument by reference involves passing a variable reference to a function, therefore giving it direct access to change that variable. These two object passing methods have also been recognized in process modeling. Object passing by reference and by value are described as workflow data patterns by Russel et al [Russell et al., 2004, Russell et al., 2005], who also point out the importance of both of these object passing methods.

We identify the following factors that influence the choice of the object passing method.

Pass-by-reference is suitable under one or more of the following conditions:

- objects can be referenced and reside in a storage location mutually accessible by actions in the process model;
- objects cannot be copied, or passing objects by value is an overhead;
- some actions should interleave, thus updating objects in an undeterminate order.

Pass-by-value is suitable under one or more of the following conditions:

- there is no mutually accessible storage between the actions in the process model;
- objects can be copied and reconciliation of object copies is possible after concurrent actions have completed;
- true concurrency of actions is required and the effects of each action on the objects need to be kept separately.

Since both methods of passing objects in a process model are valuable under different conditions, prescribing only one of these methods for object passing could impose a significant restriction on the modeler. On the other hand, allowing for both methods without providing an explicit modeling element to distinguish between the two in a model - as it is currently done in many process modeling languages like UML Activity Diagrams and BPMN - can lead to misinterpretations. For example, UML Activity Diagrams use object tokens to explain data flow in a process model, without precisely defining whether an object token corresponds to an actual object or only to an object reference.

We define a data-flow semantics that specifies the actual objects for each execution state of a process model instead of using object tokens. Since repository data flow naturally represents actions reading and writing objects to and from a shared storage, we assign a pass-by-reference semantics to this type of data flow. Figure 3.11(a) illustrates that in repository data flow, actions with data inputs and outputs of the same object type manipulate the same object[3]. To avoid the obvious problem of inconsistent state updates, we assume that actions that update an object gain exclusive write-access to that object by

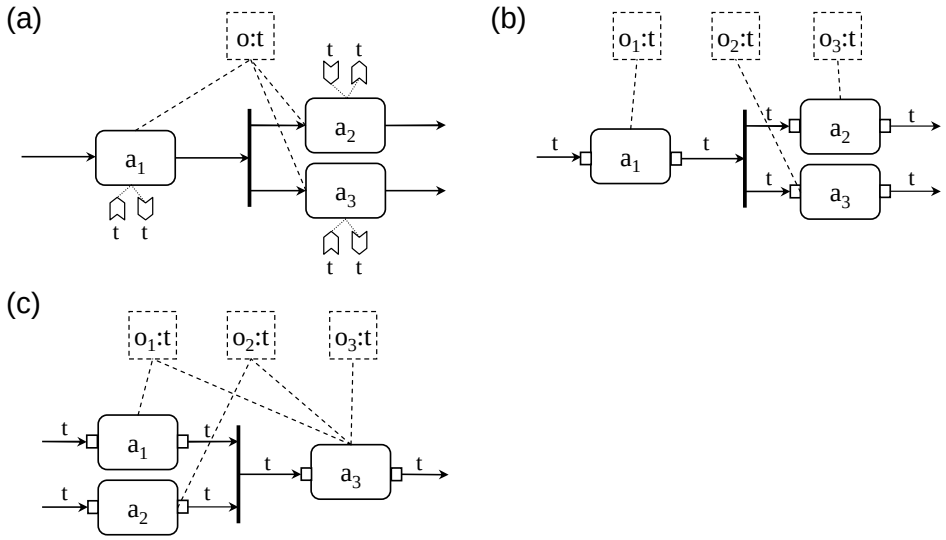


Figure 3.11: Objects in repository and routed data flow

locking it at the time they begin to execute and unlocking it when they complete the execution.

On the other hand, we interpret routed data flow to have a pass-by-value semantics. In this type of data flow, forks with typed edges make physical copies of the corresponding objects. This is illustrated in Figure 3.11(b), where the fork copies object $o_1 : t$ to produce objects $o_2 : t$ and $o_3 : t$. Hence, each of the actions a_1, a_2, a_3 manipulates a different object.

In routed data flow, joins with typed edges are responsible for synchronizing control flow only and do not themselves represent reconciliation of object copies, since reconciliation requires a custom logic. A join with typed edges is always followed by an action that reconciles the object copies. This is illustrated in Figure 3.11(c), where action a_3 reconciles objects $o_1 : t$ and $o_2 : t$ to produce object $o_3 : t$.

Figure 3.12 shows extracts from two process models for contract reviewing that use different methods of object passing. In both, after an insurance contract for a client has been drafted by the policy specialist ($Draft\ Contract$), the contract has to be reviewed by the claims specialist ($Review\ by\ CS$) and the legal expert ($Review\ by\ LE$). Revisions to the draft made by both reviewers need to be incorporated into the contract. The revision process is repeated until both, the claims specialist and the legal expert, approve the contract. Note that both of these process model extracts have a partial state specification, which is completed according to the default state specification assignment (see Definition 15). For conciseness, only one edge connected to the fork and join is explicitly marked with an object type in Figure 3.12(b), since the types of all edges connected to such nodes are the same by Definition 8.

In Figure 3.12(a), both reviewers have access to the same $Contract$ object and perform their reviews in an arbitrary order such that the second reviewer sees the outcome of the first review. The second reviewer can therefore use the information about the first review to influence his/her review. For example, if the second reviewer receives the $Contract$ in state $RevisionRequired$, he/she may even decide not to add additional comments and to simply pass the $Contract$ on in state $RevisionRequired$. Dependency states sets are specified for actions $Review\ by\ CS$ and $Review\ by\ LE$ to ensure that if

[3]Graphical elements indicated with dotted lines are added for illustration purposes only and are not part of the process model.

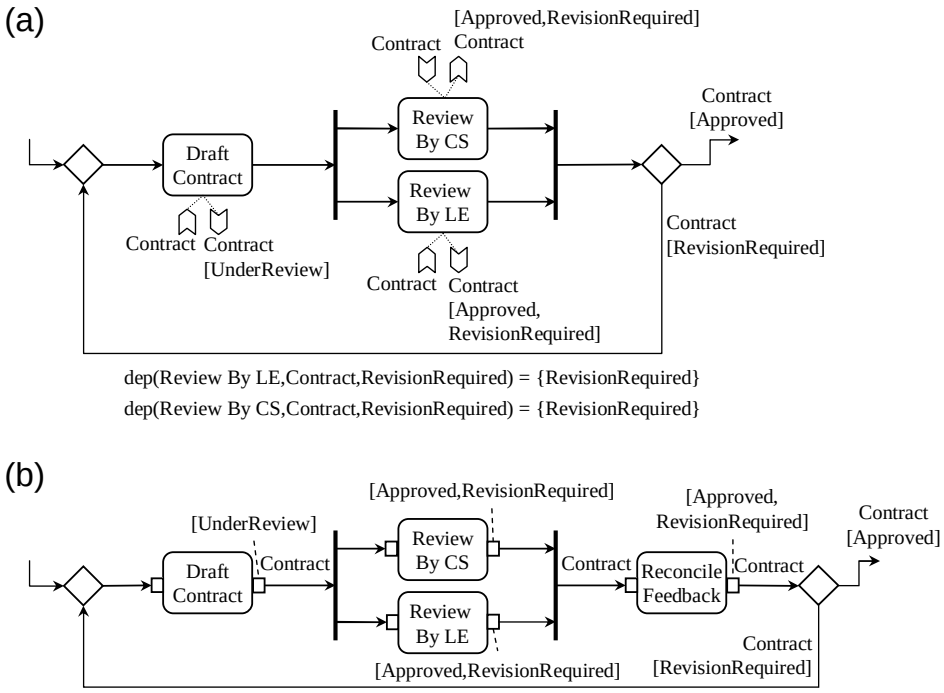


Figure 3.12: Object passing examples

the second reviewer receives a $Contract$ in state $RevisionRequired$, he/she cannot change its state to $Approved$, as this would result in the revisions required by the first reviewer being ignored.

In Figure 3.12(b), two copies of the $Contract$ object are created by the fork and passed to the review actions. The reviews are therefore performed concurrently and independently from each other. Reconciliation of the two copies requires a non-trivial procedure, e.g. consider two binary documents that have been adapted and adorned with comments independently. For this reason, the join is followed by the $Reconcile\ Feedback$ action, which reconciles the copies to produce one merged $Contract$ object.

The semantics of object manipulation and object passing is formalized next.

3.2.4 Extending Semantics with Data Flow and Object States

The control-flow semantics given by Definition 3 is extended to take into account data flow and a state specification associated with a given workflow graph. This extension involves adding new pre-conditions and post-conditions to the execution rules for each type of node that can occur in a workflow graph. Additionally, several notions related to the execution semantics presented in Section 3.2.1 are augmented.

First of all, the notion of an execution state is extended to include a set of objects that exist at that particular point in time. We assume a *universal object set*, from which new objects can be added to the object set of an execution state.

Definition 16 (Universal object set)**.** *Let a set of object types T be given. We denote the* universal object set *as $\mathcal{O}$, where each object $o \in \mathcal{O}$ is associated with an object type $t \in T$ using the mapping $type : \mathcal{O} \rightarrow T$.*

Each object in a particular execution state is also associated with a state. Once again, note the terminology: an "execution state" of a workflow graph vs. a "state" of an object. The assignment of states to objects can change from one execution state to another.

Definition 17 (Execution state: extended with data flow and object state)**.** *Let a workflow graph $G = (N, E)$ with repository or routed data flow using a set of object types T and a state specification for G be given. An* execution state *of G is represented by (w, O, s), where w is the mapping of tokens to edges as defined in Definition 2, $O \subseteq \mathcal{O}$ is a set of objects, and $s : O \to S_T$ is a mapping that assigns an object to its current state.*

In the following, we present the semantics for different data-flow types separately.

Semantics for Repository Data Flow

The execution semantics of a workflow graph with repository data flow comprises four rules for the execution of different types of nodes.

Definition 18 (Repository data-flow semantics)**.** *Let a workflow graph $G = (N, E)$ with repository data flow be given. Let (w, O, s) and (w', O', s') be execution states of G and $n \in N$ be a node that is neither a start nor a stop node. The execution of n changes (w, O, s) to (w', O', s'), written $(w, O, s) \xrightarrow[rep]{n} (w', O', s')$, if one of the following holds:*

- $(w, O, s) \xrightarrow[rep,act]{n} (w', O', s')$ *(Definition 20);*
- $(w, O, s) \xrightarrow[rep,f/j]{n} (w', O', s')$ *(Definition 21);*
- $(w, O, s) \xrightarrow[rep,dec]{n} (w', O', s')$ *(Definition 23);*
- $(w, O, s) \xrightarrow[rep,mer]{n} (w', O', s')$ *(Definition 24).*

The execution of an action can create new objects, as informally described in Section 3.2.3. We introduce the following auxiliary definition to specify the set of *new objects* created by a given action.

Definition 19 (New objects: repository data flow)**.** *Let a workflow graph $G = (N, E)$ with repository data flow and the universal object set $\mathcal{O}$ be given. We define a function $new_{rep} : N \to \mathcal{P}(\mathcal{O})$ to map a node to a set of* new objects *that this node creates. Given a node n, the following conditions hold:*

- $\forall t \in dataout(n) \setminus datain(n).\ \exists! o \in new_{rep}(n).\ type(o) = t$
- $\forall o \in new_{rep}(n).\ \exists t \in dataout(n) \setminus datain(n).\ type(o) = t$

According to Definition 19, $new_{rep}(n)$ for a given node n comprises exactly one object for each type t that is in the data outputs of n and not in the data inputs of n[4]. Since only an action can have an object type in its data outputs and not in its data inputs, $new_{rep}(n)$ is non-empty only if n is an action. The execution of an action is then defined as follows.

Definition 20 (Execution of an action: repository data-flow)**.** *Let a workflow graph $G = (N, E)$ with repository data flow be given. The* execution of an action *$n \in N$ changes an execution state (w, O, s) to another execution state (w', O', s'), written $(w, O, s) \xrightarrow[rep,act]{n} (w', O', s')$, under the following pre- and post-conditions:*

[4] $\exists!$ is the unique existential quantifier.

- *Pre 20.1: there is at least one token on the incoming edge of n in w:*

$$\forall e \in in(n).\ w(e) > 0$$

- *Pre 20.2: objects required as inputs by n exist in O and are in accepted states:*

$$\forall t \in datain(n).\ \exists o \in O.\ type(o) = t \wedge s(o) \in acpt(n, t)$$

- *Post 20.1: in w', one token is removed from the incoming edge of n and one token is added to the outgoing edge of n:*

$$w'(e) = \begin{cases} w(e) - 1 & e \in in(n) \\ w(e) + 1 & e \in out(n) \\ w(e) & \textit{otherwise.} \end{cases}$$

- *Post 20.2: O' comprises object sets O_{old}, O_{upd} and O_{new}, where O_{old} consists of unchanged objects in O, O_{upd} consists of objects in O with their states updated and O_{new} contains new objects created by n:*

$$\begin{aligned} O' &= O_{old} \cup O_{upd} \cup O_{new} \\ O_{old} &= \{o \in O \mid type(o) \notin datain(n) \cap dataout(n) \wedge type(o) \notin dataout(n) \setminus datain(n)\} \\ &\quad \textit{and } \forall o \in O_{old}.\ s'(o) = s(o) \\ O_{upd} &= \{o \in O \mid type(o) \in datain(n) \cap dataout(n)\} \textit{ and } \forall o \in O_{upd}.\ s'(o) \in dep(n, t, s(o)) \\ O_{new} &= new_{rep}(n) \textit{ and } \forall o \in O_{new}.\ s'(o) \in prod(n, type(o)) \end{aligned}$$

The pre- and post-conditions concerning the token mappings w and w' remain the same as in Definition 3, except here they are specified more explicitly. Pre 20.2 and Post 20.2 express the object manipulation semantics of *create*, *read*, *update* and *no impact*, informally described in Section 3.2.3. Pre 20.2 ensures that an action can only execute if the objects that it requires as input exist and are in accepted states (*read* and *update*). The post-condition Post 20.2 specifies how the object set and the state mapping are changed as a result of an action execution, distinguishing between O_{old} (*no impact*), O_{upd} (*update*) and O_{new} (*create*). The state of objects in O_{upd} and O_{new} is non-deterministically assigned to one of the states in $dep(n, t, s(o))$ and $prod(n, type(o))$, respectively.

According to the defined execution semantics for actions, there can only be one object of the same type in a given execution state. As illustrated in Figure 3.13, an action that creates an object of a particular type always replaces an object of that type if such an object exists. After action a_1 executes, the object o_1 of type t is created as illustrated in (a). After action a_2 executes, the object o_1 is replaced by the object o_2 of type t. By Post 20.2, o_2 is an element of O_{new} and o_1 is not included in O_{old}, because $type(o_1) \in dataout(a_2) \setminus datain(a_2)$.

The defined action execution is in accordance with the pass-by-reference semantics associated with repository data flow. Each occurrence of an object type in a process model essentially represents different references to the same object. Hence, each action that matches the *create* object manipulation operation for a given object type reassigns the referenced object.

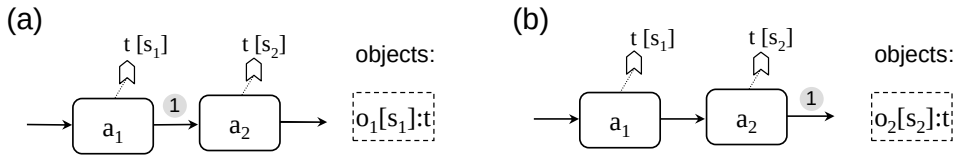


Figure 3.13: Multiple actions that create objects of the same type

The execution of a fork or a join does not affect the object set or the state mapping of an execution state, as defined below.

Definition 21 (Execution of a fork or a join: repository data-flow)**.** *Let a workflow graph $G = (N, E)$ with repository data flow be given. The* execution of a fork or a join *$n \in N$ changes an execution state (w, O, s) to another execution state (w', O', s'), written $(w, O, s) \xrightarrow[rep,f/j]{n} (w', O', s')$, under the following pre- and post-conditions:*

- *Pre 21.1: there is at least one token on each incoming edge of n in w:*

$$\forall e \in in(n).\ w(e) > 0$$

- *Post 21.1: in w', one token is removed from every incoming edge of n and one token is added to every outgoing edge of n:*

$$w'(e) = \begin{cases} w(e) - 1 & e \in in(n) \\ w(e) + 1 & e \in out(n) \\ w(e) & \textit{otherwise.} \end{cases}$$

- *Post 21.2: object set O and mapping s are unchanged:*

$$O' = O \textit{ and } \forall o \in O'.\ s'(o) = s(o)$$

Decision execution is affected by the edge conditions, as informally described in Section 3.2.3. We introduce the following auxiliary definition of an *activated edge* to refer to an outgoing edge of a decision that has all its associated edge conditions satisfied.

Definition 22 (Activated edge)**.** *Let a workflow graph $G = (N, E)$ with repository or routed data flow be given. Given a decision $n \in N$ and an execution state (w, O, s) of G, an outgoing edge $e \in out(n)$ of a decision n is said to be an* activated edge *in (w, O, s) if and only if $\forall o \in O.\ s(o) \in cond(e, type(o))$. We denote the set of all activated edges of a decision n in (w, O, s) as $activ(n, (w, O, s))$.*

The execution of a decision is then defined as follows.

Definition 23 (Execution of a decision: repository data-flow)**.** *Let a workflow graph $G = (N, E)$ with repository data flow be given. The* execution of a decision *$n \in N$ changes an execution state (w, O, s) to another execution state (w', O', s'), written $(w, O, s) \xrightarrow[rep,dec]{n} (w', O', s')$, under the following pre- and post-conditions:*

- *Pre 23.1: there is at least one token on the incoming edge of n in w:*

$$\forall e \in in(n).\ w(e) > 0$$

- *Pre 23.2: n has at least one activated outgoing edge in (w, O, s):*

$$|activ(n, (w, O, s))| > 0$$

- *Post 23.1: in w', one token is removed from the incoming edge of n and one token is added to one of the activated outgoing edges of n:*

$$w'(e) = \begin{cases} w(e) - 1 & e \in in(n) \\ w(e) + 1 & e = e', \text{ where } e' \in activ(n, (w, O, s)) \\ w(e) & \text{otherwise.} \end{cases}$$

- *Post 23.2: object set O and mapping s are unchanged:*

$$O' = O \text{ and } \forall o \in O'.\ s'(o) = s(o)$$

The original semantics of decision execution presented in Definition 3 is extended with Pre 23.2, which ensures that the given decision has at least one activated edge. Post 23.1 specifies that only an activated edge can receive a token as a result of a decision execution.

The execution of a merge does not affect the object set or the state mapping of an execution state.

Definition 24 (Execution of a merge: repository data-flow)**.** *Let a workflow graph $G = (N, E)$ with repository data flow be given. The* execution of a merge $n \in N$ *changes an execution state (w, O, s) to another execution state (w', O', s'), written $(w, O, s) \xrightarrow[rep,mer]{n} (w', O', s')$, under the following pre- and post-conditions:*

- *Pre 24.1: there is at least one token on one of the incoming edge of n in w:*

$$\exists e \in in(n).\ w(e) > 0$$

- *Post 24.1: in w', one token is removed from an incoming edge of n and one token is added to the outgoing edge of n:*

$$w'(e) = \begin{cases} w(e) - 1 & e = e', \text{ where } e' \in in(n) \wedge w(e') > 0 \\ w(e) + 1 & e \in out(n) \\ w(e) & \text{otherwise.} \end{cases}$$

- *Post 24.2: object set O and mapping s are unchanged:*

$$O' = O \text{ and } \forall o \in O'.\ s'(o) = s(o)$$

This completes the execution semantics for repository data flow. In the following, we define the semantics for routed data flow in a similar style.

Semantics for Routed Data Flow

The execution semantics of a workflow graph with routed data flow comprises five rules for the execution of different types of nodes. As described in Section 3.2.3, we assume that in a workflow graph with routed data flow, a join with typed edges is always followed by an action that represents the reconciliation of objects.

Definition 25 (Routed data-flow semantics)**.** *Let a workflow graph $G = (N, E)$ with routed data flow be given. Let (w, O, s) and (w', O', s') be execution states of G and $n \in N$ be a node that is neither a start nor a stop node. The execution of n changes (w, O, s) to (w', O', s'), written $(w, O, s) \xrightarrow[rout]{n} (w', O', s')$, if one of the following holds:*

- $(w, O, s) \xrightarrow[rout,act]{n} (w', O', s')$ *(Definition 30);*
- $(w, O, s) \xrightarrow[rout,fork]{n} (w', O', s')$ *(Definition 31);*
- $(w, O, s) \xrightarrow[rout,join]{n} (w', O', s')$ *(Definition 33);*
- $(w, O, s) \xrightarrow[rout,dec]{n} (w', O', s')$ *(Definition 34);*
- $(w, O, s) \xrightarrow[rout,mer]{n} (w', O', s')$ *(Definition 35).*

In routed data flow, objects of the same type need to be distinguished, since a node should only have access to the objects received via its incoming edges and not to all the existing objects of the same type as the node's incoming edges. We introduce the notion of an *object label* to identify each object by the edge that was last traversed by the object.

Definition 26 (Object label)**.** *Let a workflow graph $G = (N, E)$ with routed data flow and an execution state (w, O, s) of G be given. Each object $o \in O$ is associated with an edge that is referred to as its* object label *via the mapping $label : O \to E$. Given an object $o \in O$ and an edge $e \in E$ such that $label(o) = e$, we write o_e.*

Figure 3.14(a) illustrates that an action a with an incoming edge e_1 and an outgoing edge e_2 of type t receives an object o_{e_1} and produces an object o_{e_2}. Any other object o of type t that is not labeled with either e_1 or e_2 is not manipulated by a. This example represents an update operation, as discussed in Section 3.2.3. In routed data flow, an update involves a removing object o_{e_1} from the set of existing objects and adding a new object o_{e_2} to that set. In accordance with the pass-by-value semantics assigned to routed data flow, o_{e_2} represents a copy of o_{e_1}. Figure 3.14(b) additionally shows that actions following a join may receive more than one object labeled with the same edge.

The notion of new objects is augmented to take into account the object labels.

Definition 27 (New objects: routed data flow)**.** *Let a workflow graph $G = (N, E)$ with routed data flow and the universal object set $\mathcal{O}$ be given. We define a function $new_{rout} : N \to \mathcal{P}(\mathcal{O})$ to map a node to a set of* new objects *that this node creates. Given a node n, the following conditions hold:*

- $\forall e \in out(n).\ type(e) \in dataout(n) \setminus datain(n) \Rightarrow$
  $\exists! o_e \in new_{rout}(n).\ type(o_e) = type(e)$
- $\forall o_e \in new_{rep}(n).\ \exists t \in dataout(n) \setminus datain(n).\ type(o_e) = t$

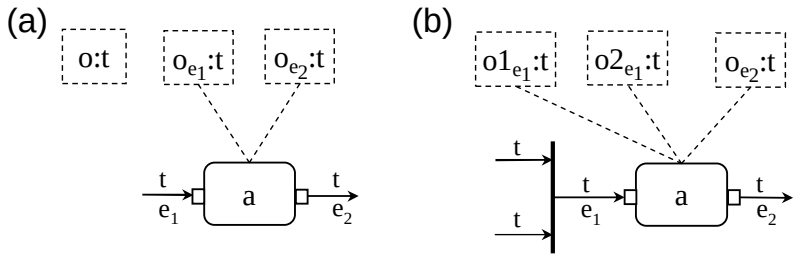


Figure 3.14: Object labels

We introduce the concept of *replacement objects* to specify the objects added by an action or another node n to the set of existing objects as a replacement for the objects removed by n.

Definition 28 (Replacement objects)**.** *Let a workflow graph $G = (N, E)$ with routed data flow and the universal object set $\mathcal{O}$ be given. We define a function $repl : N \to \mathcal{P}(\mathcal{O})$ to map a node to a set of* replacement objects *that this node creates as a replacement for the objects it receives as input. Given a node n, the following conditions hold:*

- $\forall e \in out(n).\ type(e) \in datain(n) \cap dataout(n) \Rightarrow \exists! o_e \in repl(n).\ type(o_e) = type(e)$
- $\forall o_e \in repl(n).\ \exists t \in datain(n) \cap dataout(n).\ type(o_e) = t$

Furthermore, we introduce the concept of *prior objects* to refer to the objects replaced with a given object o as a result of the execution of a given node n.

Definition 29 (Prior objects)**.** *Let a workflow graph $G = (N, E)$ with routed data flow and the universal object set $\mathcal{O}$ be given. We define a function $prior : N \times \mathcal{P}(\mathcal{O}) \times \mathcal{O} \to \mathcal{P}(\mathcal{O})$ to map a node $n \in N$, an object set $O \subseteq \mathcal{P}(\mathcal{O})$ and an object $o \in \mathcal{O}$ to a set of* prior objects *in O that n replaces with o. Given a node $n \in N$, an object set $O \subseteq \mathcal{P}(\mathcal{O})$ and an object $o \in \mathcal{O}$ such that $o \in repl(n)$, $prior(n, O, o) = \{o_e \in O \mid e \in in(n) \wedge type(e) = type(o)\}$.*

The execution of an action is then defined as follows.

Definition 30 (Execution of an action: routed data-flow)**.** *Let a workflow graph $G = (N, E)$ with routed data flow be given. The* execution of an action $n \in N$ *changes an execution state (w, O, s) to another execution state (w', O', s'), written as $(w, O, s) \xrightarrow[rout,act]{n} (w', O', s')$, under the following pre- and post-conditions:*

- *Pre 30.1: there is at least one token on each incoming edge of n in w:*

$$\forall e \in in(n).\ w(e) > 0$$

- *Pre 30.2: all objects in O required as inputs by n are in accepted states:*

$$\forall e \in in(n).\ \forall o_e \in O.\ s(o_e) \in acpt(n, type(o_e))$$

- *Post 30.1: in w', one token is removed from every incoming edge of n and one token is added to every outgoing edge of n:*

$$w'(e) = \begin{cases} w(e) - 1 & e \in in(n) \\ w(e) + 1 & e \in out(n) \\ w(e) & \textit{otherwise.} \end{cases}$$

- *Post 30.2: object set O' comprises object sets O_{old}, O_{upd} and O_{new}, where O_{old} consists of unchanged objects in O, O_{upd} comprises objects created by n to replace objects in $O \setminus O_{old}$ and O_{new} contains completely new objects created by n:*

$$\begin{aligned} O' &= O_{old} \cup O_{upd} \cup O_{new} \\ O_{old} &= \{o_e \in O \mid e \notin in(n) \vee type(e) \notin dataout(n)\} \text{ and } \forall o_e \in O_{old}.\ s'(o_e) = s(o_e) \\ O_{upd} &= repl(n) \text{ and } \forall o_e \in O_{upd}.\ s'(o_e) \in \textstyle\bigcup_{o \in prior(n,O,o_e)} dep(n, type(e), s(o)) \\ O_{new} &= new_{rout}(n) \text{ and } \forall o_e \in O_{new}.\ s'(o_e) \in prod(n, type(e)) \end{aligned}$$

The pre- and post-conditions for action execution are defined similarly to those for repository data flow, with the only difference being the definition of Post 30.2. Objects in O_{old} (*no impact*) comprise objects labeled with edges other than the incoming edges of a given action and those labeled with edges of type t that is not in the data outputs of the given action. Objects in O_{upd} (*update*) comprise replacement objects of the given action. The update of the state mapping for these objects can be illustrated using the example shown in Figure 3.14(b) where action a replaces objects $o1_{e_1}$ and $o2_{e_1}$ with object o_{e_2}. Assuming that $s(o1_{e_1}) = s_1$ and $s(o2_{e_1}) = s_2$, the state mapping for object o_{e_2} after the execution of a must be $s'(o_{e_2}) = s_3$ such that $s_3 \in dep(a, t, s_1) \cup dep(a, t, s_2)$. Objects in O_{new} (*create*) are defined similarly to those for repository data flow.

The semantics given in Definition 30 also applies to actions in relaxed workflow graphs, i.e. actions that have multiple incoming or outgoing edges (cf. Definition 9).

The execution of a fork node is defined as follows.

Definition 31 (Execution of a fork: routed data-flow)**.** *Let a workflow graph $G = (N, E)$ with routed data flow be given. The* execution of a fork $n \in N$ *changes an execution state (w, O, s) to another execution state (w', O', s'), written $(w, O, s) \xrightarrow[rout,fork]{n} (w', O', s')$, under the following pre- and post-conditions:*

- *Pre 31.1: there is at least one token on the incoming edge of n in w:*

$$\forall e \in in(n).\ w(e) > 0$$

- *Post 31.1: in w', one token is removed from the incoming edge of n and one token is added to every outgoing edge of n:*

$$w'(e) = \begin{cases} w(e) - 1 & e \in in(n) \\ w(e) + 1 & e \in out(n) \\ w(e) & \textit{otherwise.} \end{cases}$$

- *Post 31.2: object set O' comprises object sets O_{old} and O_{repl}, where O_{old} contains objects in O that remain unchanged and O_{repl} comprises objects created by n to replace objects in $O \setminus O_{old}$:*

$$\begin{aligned} O' &= O_{old} \cup O_{repl} \\ O_{old} &= \{o_e \in O \mid e \notin in(n)\} \text{ and } \forall o_e \in O_{upd}.\ s'(o_e) = s(o_e) \\ O_{repl} &= repl(n) \text{ and } \forall o_e \in O_{repl}.\ s'(o_e) \in \textstyle\bigcup_{o \in prior(n,O,o_e)} s(o) \end{aligned}$$

The semantics of a fork execution remains the same as in Definition 3, except that a new post-condition Post 31.2 is introduced. According to Post 31.2, the execution of a fork removes objects labeled with its incoming edge (these objects are not included in O_{old}) and adds a new object for each of its outgoing edges (O_{repl}). Note that there can only be one prior object for the replacement objects of a fork, since multiple objects labeled with the same edge are always reconciled into one object by actions. As a result, according to Post 31.2, the state of the replacement objects of a fork is the same as the state of their prior object.

The remaining join, decision and merge do not remove or add objects during their execution, but rather relabel the existing objects with different edges. We introduce the following notion of a *relabeled object set* to refer to the set of objects relabeled by a given node.

Definition 32 (Relabeled object set)**.** *Let a workflow graph $G = (N, E)$ with routed data flow and the universal object set $\mathcal{O}$ be given. We define a function $relabel : \mathcal{P}(\mathcal{O}) \times E \times E \rightarrow \mathcal{P}(\mathcal{O})$ to map an object set $O \subseteq \mathcal{O}$ and edges $e, e' \in E$ to a* relabeled object set *such that each object $o_e \in O$ is relabeled to $o_{e'}$ in $relabel(O, e, e')$.*

The execution of a join, a decision and a merge is then defined as follows.

Definition 33 (Execution of a join: routed data-flow)**.** *Let a workflow graph $G = (N, E)$ with routed data flow be given. The* execution of a join *$n \in N$ changes an execution state (w, O, s) to another execution state (w', O', s'), written $(w, O, s) \xrightarrow[rout,join]{n} (w', O', s')$, under the following pre- and post-conditions:*

- *Pre 33.1: there is at least one token on each incoming edge of n in w:*

$$\forall e \in in(n).\ w(e) > 0$$

- *Post 33.1: in w', one token is removed from each incoming edge of n and one token is added to the outgoing edge of n:*

$$w'(e) = \begin{cases} w(e) - 1 & e \in in(n) \\ w(e) + 1 & e \in out(n) \\ w(e) & \textit{otherwise.} \end{cases}$$

- *Post 33.2: object set O is unchanged, except that each object labeled with an incoming edge of n is relabeled with the outgoing edge of n:*

$$O' = \{o \in relabel(O, e, e') \mid e \in in(n) \wedge e' \in out(n)\} \textit{ and } \forall o \in O'.\ s'(o) = s(o)$$

Definition 34 (Execution of a decision: routed data-flow)**.** *Let a workflow graph $G = (N, E)$ with routed data flow be given. The* execution of a decision *$n \in N$ changes an execution state (w, O, s) to another execution state (w', O', s'), written $(w, O, s) \xrightarrow[rout,dec]{n} (w', O', s')$, under the following pre- and post-conditions:*

- *Pre 34.1: there is at least one token on the incoming edge of n in w:*

$$\forall e \in in(n).\ w(e) > 0$$

- *Pre 34.2: n has at least one activated outgoing edge in (w, O, s):*

$$|activ(n, (w, O, s))| > 0$$

- *Post 34.1: in w', one token is removed from the incoming edge of n and one token is added to one of the activated outgoing edges of n:*

$$w'(e) = \begin{cases} w(e) - 1 & e \in in(n) \\ w(e) + 1 & e = e', \text{ where } e' \in activ(n, (w, O, s)) \\ w(e) & \text{otherwise.} \end{cases}$$

- *Post 34.2: object set O is unchanged, except that each object labeled with the incoming edge of n is relabeled with the outgoing edge of n that receives an additional token by Post 34.1:*

$$\begin{aligned} O' = &\{o \in relabel(O, e, e') \mid e \in in(n)\} \text{ and } \forall o \in O'.\ s'(o) = s(o), \\ &\text{where } e' \in activ(n, (w, O, s)) \text{ s.t. } w'(e') = w(e') + 1 \end{aligned}$$

Definition 35 (Execution of a merge: routed data-flow)**.** *Let a workflow graph $G = (N, E)$ with routed data flow be given. The* execution of a merge $n \in N$ *changes an execution state (w, O, s) to another execution state (w', O', s'), written $(w, O, s) \xrightarrow[rout,mer]{n} (w', O', s')$, under the following pre- and post-conditions:*

- *Pre 35.1: there is at least one token on one of the incoming edges of n in w:*

$$\exists e \in in(n).\ w(e) > 0$$

- *Post 35.1: in w', one token is removed from an incoming edge of n and one token is added to the outgoing edge of n:*

$$w'(e) = \begin{cases} w(e) - 1 & e = e', \text{ where } e' \in in(n) \wedge w(e') > 0 \\ w(e) + 1 & e \in out(n) \\ w(e) & \text{otherwise.} \end{cases}$$

- *Post 35.2: object set O is unchanged, except that each object labeled with the incoming edge of n, from which a token is removed by Post 35.1, is relabeled with the outgoing edge of n:*

$$\begin{aligned} O' = &\{o \in relabel(O, e, e') \mid e' \in out(n)\} \text{ and } \forall o \in O'.\ s'(o) = s(o), \\ &\text{where } e \in in(n) \text{ s.t. } w'(e) = w(e) - 1 \end{aligned}$$

This completes the execution semantics for routed data flow. In the following, we present the necessary extensions to some other notions related to the execution of process models.

Extending Common Semantic Definitions

We extend the definition of the initial execution state, in which only the outgoing edge of the start node has a token, to be associated with an empty set of objects.

Definition 36 (Initial execution state: extended with data flow and object state)**.** *Given a workflow graph G with repository or routed data flow, its* initial execution state *is the execution state (w, O, s) such that w has exactly one token on the outgoing edge of the start node and no tokens elsewhere, and $O = \emptyset$.*

It would also be possible to pre-populate the set O with objects that exist before the process begins execution to capture process data inputs, however we assume it to be empty for the clarity of the theory developed later in this dissertation. We maintain the definition of the terminal execution state introduced in Section 3.2.1.

The definitions of an activated node, an execution sequence, and a reachable execution state are extended as follows.

Definition 37 (Activated node, execution sequence: extended with data flow and object state, control-flow activated node)**.** *A node n is said to be* activated *in an execution state (w, O, s) if there exists another execution state (w', O', s') such that $(w, O, s) \xrightarrow{n} (w', O', s')$. Additionally, we say that a node n is* control-flow activated *in an execution state (w, O, s) if there exists an execution state w' such that $w \xrightarrow{n} w'$ according to the control-flow semantics in Definition 3. A sequence of node executions $(w_0, O_0, s_0) \xrightarrow{n_1} (w_1, O_1, s_1) \dots (w_{k-1}, O_{k-1}, s_{k-1}) \xrightarrow{n_k} (w_k, O_k, s_k)$ is called an* execution sequence*.*

Definition 38 (Reachable execution state: extended with data flow and object state)**.** *An execution state (w', O', s') is* reachable *from an execution state (w, O, s), denoted $(w, O, s) \xrightarrow{*} (w', O', s')$, if there exists a (possibly empty) finite execution sequence $(w_0, O_0, s_0) \xrightarrow{n_1} (w_1, O_1, s_1) \dots (w_{k-1}, O_{k-1}, s_{k-1}) \xrightarrow{n_k} (w_k, O_k, s_k)$ such that $(w_0, O_0, s_0) = (w, O, s)$ and $(w_k, O_k, s_k) = (w', O', s')$.*

We next illustrate the extended semantics by means of an example. Figure 3.15 shows two versions of the original claims handling process model introduced in Figure 3.1, where names of actions, object types and states are abbreviated and edges are uniquely labeled. The original process model in Figure 3.1 uses routed data flow, however the join in the original process model is not followed by an action, as required by routed data flow. Therefore, we add a $Confirm\ Evaluation$ (CE) action after the join to reconcile the claim copies, as shown in Figure 3.15(a). The process model shown in (b) replaces the routed data flow by repository data flow in the original claims handling process model.

An example execution sequence of the process model in Figure 3.15(a) is given in Table 3.1. For each execution state, the edges containing tokens are given in column w, the existing objects are given in column O and the state mapping is given in column s. The type of objects is not explicitly given, since all objects are of type $Claim$ in this example.

In Table 3.1, it can be seen that no objects exist in the initial execution state, two objects exist in four execution states, while only one object exists in all other execution states. In this execution sequence, the *CFF* ($Check\ For\ Fraud$) action updates the state of a $Claim$ copy to Fr ($Fraud$) (execution state (w_4, O_4, s_4)) and the EC ($Evaluate\ Claim$) action updates the state of the other $Claim$ copy to Gr ($Granted$) (execution state (w_5, O_5, s_5)). Subsequently, the $Confirm\ Evaluation$ (CE) action produces a reconciled $Claim$ object in state $Rejected$ (Rj) (execution state (w_7, O_7, s_7)). We can imagine that

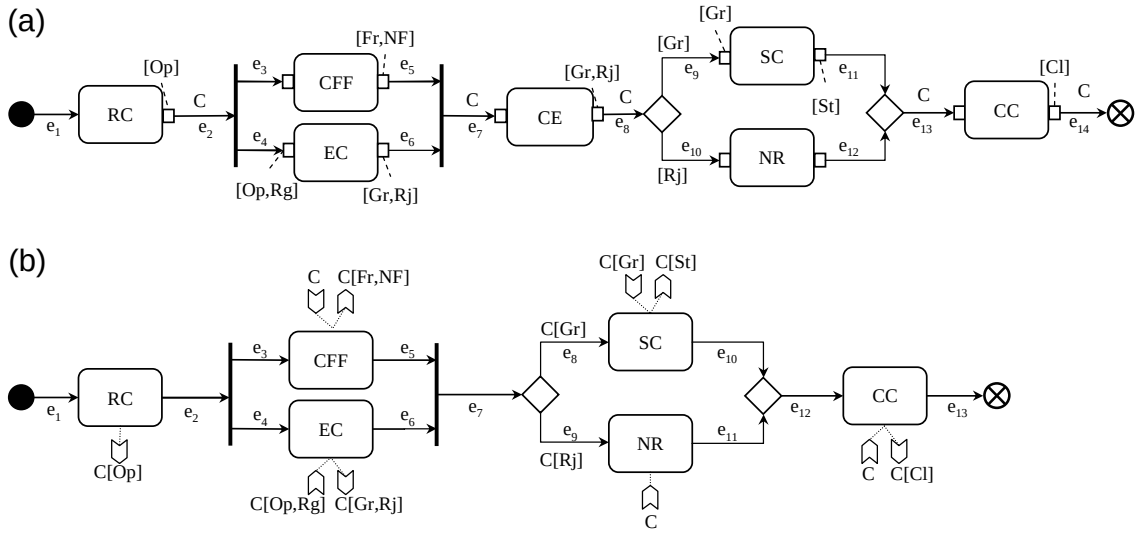


Figure 3.15: Executing process models with different types of data flow

Table 3.1: Example execution sequence for process model in Figure 3.15(a)

Exec. state	w	O	s
(w_1, O_1, s_1)	$w_1(e_1) = 1$	$\{\}$	
(w_2, O_2, s_2)	$w_2(e_2) = 1$	$\{o_{e_2}\}$	$s_2(o_{e_2}) = Op$
(w_3, O_3, s_3)	$w_3(e_3) = 1$ and $w_3(e_4) = 1$	$\{o_{e_3}, o_{e_4}\}$	$s_3(o_{e_3}) = s_3(o_{e_4}) = Op$
(w_4, O_4, s_4)	$w_4(e_5) = 1$ and $w_4(e_4) = 1$	$\{o_{e_5}, o_{e_4}\}$	$s_4(o_{e_5}) = Fr$, $s_4(o_{e_4}) = Op$
(w_5, O_5, s_5)	$w_5(e_5) = 1$ and $w_5(e_6) = 1$	$\{o_{e_5}, o_{e_6}\}$	$s_5(o_{e_5}) = Fr$, $s_5(o_{e_4}) = Gr$
(w_6, O_6, s_6)	$w_6(e_7) = 1$	$\{o1_{e_7}, o2_{e_7}\}$	$s_6(o1_{e_7}) = Fr$, $s_6(o2_{e_4}) = Gr$
(w_7, O_7, s_7)	$w_7(e_8) = 1$	$\{o_{e_8}\}$	$s_7(o_{e_8}) = Rj$
(w_8, O_8, s_8)	$w_8(e_{10}) = 1$	$\{o_{e_{10}}\}$	$s_8(o_{e_{10}}) = Rj$
(w_9, O_9, s_9)	$w_9(e_{12}) = 1$	$\{o_{e_{12}}\}$	$s_9(o_{e_{12}}) = Rj$
(w_{10}, O_{10}, s_{10})	$w_{10}(e_{13}) = 1$	$\{o_{e_{13}}\}$	$s_{10}(o_{e_{13}}) = Rj$
(w_{11}, O_{11}, s_{11})	$w_{11}(e_{14}) = 1$	$\{o_{e_{14}}\}$	$s_{11}(o_{e_{14}}) = Cl$

the $Confirm\ Evaluation$ (CE) action reconciles the $Claim$ copies in such a way that the state of the produced $Claim$ object is $Granted$ only if one of the received $Claim$ copies is in state $Granted$ and the other is in state $NoFraud$. Capturing this type of logic goes beyond the dependency state sets used in the process models we consider here though.

Table 3.2 shows an example execution sequence of the process model in Figure 3.15(b). Only one object exists in all the shown execution states. After the execution of the RC ($Register\ Claim$) action, the fork and the CFF ($Check\ For\ Fraud$) action, no more nodes can execute (execution state (w_4, O_4, s_4)). Although edge e_4 has a token, the action EC ($Evaluate\ Claim$) cannot execute, because the $Claim$ object o is in state Fr ($Fraud$), which is not one of the accepted states defined for this action. As a result, the terminal execution state is not reached. This example illustrates that unexpected behavior, in this case a deadlock, can result from improper state specifications. In the next chapter, we come back to this topic and describe in detail the concept of a correct state specification.

We have now presented the semantics of data flow, distinguishing between routed and repository types of data flow, and object state specifications in process models. This semantics is assumed in the remainder of the dissertation. We next continue to defining

Table 3.2: Example execution sequence for process model in Figure 3.15(b)

Exec. state	w	O	s
(w_1, O_1, s_1)	$w_1(e_1) = 1$	$\{\}$	
(w_2, O_2, s_2)	$w_2(e_2) = 1$	$\{o\}$	$s_2(o) = Op$
(w_3, O_3, s_3)	$w_3(e_3) = 1$ and $w_3(e_4) = 1$	$\{o\}$	$s_3(o) = Op$
(w_4, O_4, s_4)	$w_4(e_5) = 1$ and $w_4(e_4) = 1$	$\{o\}$	$s_4(o) = Fr$

the syntax and semantics of object life cycle models.

3.3 Syntax and Semantics of an Object Life Cycle Model

In the same way as we selected a generic representation for process models, we do not focus on one specific language for object life cycle modeling, but rather consider the fundamental constructs shared by different representations. In Chapter 2, we discussed the different approaches to modeling object life cycles, which include notations based on state transition diagrams [Ebert and Engels, 1997] and state machines [Stumptner and Schrefl, 2000], Petri nets [van der Aalst and Basten, 2001] and object behavior diagrams [Kappel and Schrefl, 1991, Schrefl and Stumptner, 2002], labeled transition systems [Müller et al., 2006] and entity life histories [Jackson, 1983]. At the heart of all these approaches lie states, with distinguished initial and final states, and transitions between the states. These can be formalized as a *finite state automaton* [Hopcroft et al., 2006], which we use as the basis for defining the abstract syntax of an object life cycle model in this dissertation. Object life cycle models without concurrent states created in other representations can be easily mapped to finite state automata.

Definition 39 (Object life cycle model)**.** *Given an object type t, its* object life cycle model $OLC_t = (S, s_\alpha, s_\omega, \Sigma, \delta)$ *consists of a finite set of* states S*, where $s_\alpha \in S$ is the* initial state *and $s_\omega \in S$ is the* final state*; a finite set of* events Σ*; and a* transition function $\delta : S \times \Sigma \rightarrow \mathcal{P}(S)$*. Given $s_2 \in \delta(s_1, e)$, we write $s_1 \xrightarrow{e} s_2$. Additionally, the following conditions hold:*

- *each non-initial state $s \in S \setminus \{s_\alpha\}$ has at least one incoming transition;*
- *each non-final state $s \in S \setminus \{s_\omega\}$ has at least one outgoing transition.*

As the concrete syntax, we use the notation suggested by UML State Machines [UML, 2007a]. In the example shown in Figure 3.16, the object life cycle model for the $Claim$ object type defines seven states: the initial state and states $Open$, $Granted$, $Rejected$, $Appealed$, $Closed$ and the final state. The initial and final states of an object life cycle model are so-called pseudo-states, which serve the purpose of indicating the beginning and end of a life cycle, respectively.

In the context of this dissertation, we interpret an object life cycle model as a state evolution protocol defined for a particular object type. As automata are used to represent a language that comprises words built up from symbols of an alphabet [Hopcroft et al., 2006], an object life cycle model represents all possible state sequences of objects of a given type. In automata or language theory, the main focus is on the sequence of symbols that are associated with transitions, while the labeling of states themselves does not play an important role. In the context of object life cycle models, we

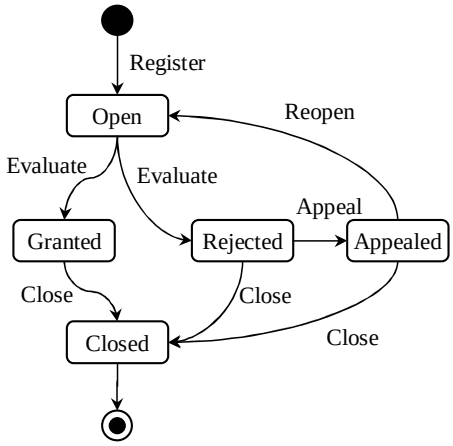


Figure 3.16: Object life cycle model

place the focus on state labels rather than the transition labels, as states are commonly standardized within an industry or at least have a unique meaning. In Figure 3.16 for example, the model is interpreted to allow $Claim$ objects to go through state sequences such as $\langle Initial, Open, Granted, Closed, Final\rangle$ and $\langle Initial, Open, Rejected, Closed, Final\rangle$. The transition labels, such as $Register$ and $Evaluate$, are not taken into account as such when considering state sequences allowed by an object life cycle model.

Since an object life cycle represents a state evolution protocol, we should be able to determine whether a given set of object state sequences conforms to the protocol. In the context of business objects, there is always a particular system that creates them and subsequently updates their state. Therefore, we can directly think of an object life cycle model to define a protocol that such a system should conform to. To get an intuitive understanding of this, we can imagine that some system X notifies an object life cycle model OLC_t of all the state changes that it induces on objects of type t to determine whether it is conformant with OLC_t, as illustrated in Figure 3.17.

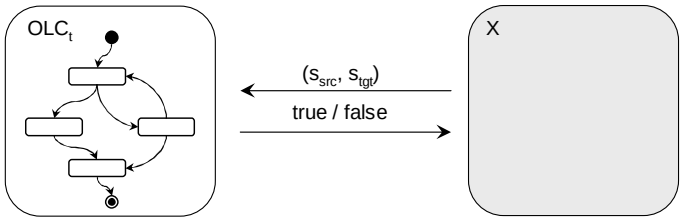


Figure 3.17: Object life cycle model as a state evolution protocol

OLC_t is originally in its initial state, and given a change state message in the form of (s_{src}, s_{tgt}) from X as input, it produces either *true* or *false* as output. The output is *true* if the current state of OLC_t is s_{src} and there is a transition from s_{src} to s_{tgt}, in which case s_{tgt} becomes the current state of OLC_t. Otherwise, the output is *false*.

Based on this semantic interpretation, the main purpose of an object life cycle model is thus to check conformance of object state changes induced by a given system. We define *object life cycle conformance* as follows.

Definition 40 (Object life cycle conformance)**.** *Let $Q = \{\langle s_{11}, ..., s_{1p}\rangle, ..., \langle s_{n1}, ..., s_{nq}\rangle\}$ be a set of state sequences generated by all possible executions of some system X. We say that*

X is conformant *with a given object life cycle model $OLC_t = (S, s_\alpha, s_\omega, \Sigma, \delta)$ if and only if the following conditions hold for every state sequence $q = \langle s_1, ..., s_r \rangle$ in Q:*

C40.1: there is a transition from s_i to s_{i+1} in OLC_t for $1 \leq i < r$;

C40.2: sequence q begins with the initial state of OLC_t, i.e. $s_1 = s_\alpha$;

C40.3: sequence q ends with the final state of OLC_t, i.e. $s_r = s_\omega$.

With regards to the object life cycle model for $Claim$ shown in Figure 3.16, a given system is not conformant if it generates state sequences such as $\langle Open, Granted, Closed \rangle$, $\langle Initial, Open, Granted, Rejected, Closed, Final \rangle$ or $\langle Initial, Open, Granted, Appealed \rangle$. A system that does not use objects of type $Claim$ at all, or one that always grants claims and thus generates only one state sequence $\langle Initial, Open, Granted, Closed, Final \rangle$, is conformant with the object life cycle model in Figure 3.16.

In some cases, conformance may not be sufficient and it may also be of importance to know whether the system at hand provides a so-called coverage of the entire object life cycle model. We therefore define *object life cycle coverage* to express the property of a system with respect to a given object life cycle model, which guarantees that all the states and transitions in the object life cycle model are covered during the possible executions of the system. This notion is comparable with the *absence of dead methods* property associated with object life cycles by van der Aalst and Basten in [van der Aalst and Basten, 2001], except that dead methods are determined by checking the behavior specified within an object life cycle model, while coverage is determined with respect to another system.

Definition 41 (Object life cycle coverage)**.** *Let $Q = \{\langle s_{11}, ..., s_{1p} \rangle, ..., \langle s_{n1}, ..., s_{nq} \rangle\}$ be a set of state sequences generated by all possible executions of some system X. We say that X provides a* coverage *of a given object life cycle model $OLC_t = (S, s_\alpha, s_\omega, \Sigma, \delta)$ if and only if the following conditions hold:*

C41.1: for each transition $s_1 \xrightarrow{e} s_2$ in OLC_t, there exists a state sequence $\langle ..., s_1, s_2, ... \rangle$ in Q where s_1 is directly followed by s_2;

C41.2: there exists a sequence $\langle s_\alpha, ... \rangle$ in Q that begins with the initial state of OLC_t;

C41.3: there exists a sequence $\langle ..., s_\omega \rangle$ in Q that ends with the final state of OLC_t.

According to the above definition, a system that always grants claims and hence generates only one state sequence $\langle Initial, Open, Granted, Closed, Final \rangle$ does not provide a coverage of the object life cycle model in Figure 3.16, but is conformant with this model.

In the remainder of the dissertation, we use object life cycle conformance and coverage as the main semantic interpretations of an object life cycle model.

3.4 Summary and Discussion

In this chapter, we defined the syntax and semantics of process and object life cycle models.

In the context of process models, we studied those modeling aspects that are not adequately described in the existing literature from the semantic point of view. These comprised the object manipulation and object passing aspects of data flow and the specification of object states in a process model. We first described the required semantics of these informally and then formalized it as an extension of an existing definition of control-flow

semantics for workflow graphs. The formalization includes two types of data-flow, namely repository and routed data flow, which we assign the pass-by-reference and pass-by-value semantics, respectively. For both types of data flow, we defined the semantics of an object state specification in a process model.

Our current definition of a process model and its data-flow semantics does not cater for the processing of object collections, e.g. having actions that require m objects of type t_1 and produce n objects of type t_2. The syntactic and semantic definitions presented in this chapter could however be extended to support this feature. For instance, this could be done by assigning multiplicities to data inputs and outputs of actions and other nodes. Since we concentrate on the fundamental aspects of process models in this dissertation, we consider such an extension to be out of our scope.

For object life cycle models, we first explained their intuitive interpretation as state evolution protocols for objects of a particular type. We then defined the semantics of an object life cycle model in terms of object life cycle conformance and coverage, which both express properties checked against the executions of a given system.

Chapter 4

Consistency

In this chapter, we address consistency of process and object life cycle models. We begin by providing a brief introduction to model consistency in Section 4.1. We then focus on intra-model consistency aspects in Section 4.2, since establishing intra-model consistency is a prerequisite for tackling the issues of inter-model consistency. In this regard, we address the correctness of process models with state specifications, as it has not yet been dealt with in the existing literature. As we do not identify any intra-model consistency issues for object life cycle models, we then proceed to defining inter-model consistency for process and object life cycle models in Section 4.3. We define process and object life cycle consistency by first identifying a common semantic domain for these two types of models. Finally, we describe how the evaluation of state specification correctness and process and object life cycle consistency is facilitated in Section 4.4.

4.1 Model Consistency

The original definition of consistency was established in classical logics (see e.g [Ebbinghaus et al., 1996]), where a theory Φ is considered consistent if and only if there is no formula ψ such that both ψ and $\neg\psi$ can be derived from Φ. Although this definition is not directly applicable to software models, the overall concept of consistency has become central to multi-view modeling and Model-Driven Engineering (MDE) [Kent, 2002, Schmidt, 2006]. Two main consistency types are distinguished: intra-model and inter-model consistency [Huzar et al., 2005] (cf. Chapter 2). Intra-model consistency is concerned with establishing that a plausible relation exists between the elements of one model. For example, a notion of intra-model consistency for UML Class Diagrams requires at least one instance of a given class diagram to exist [Wahler, 2008]. The absence of deadlocks is an example of an intra-model consistency property for behavioral models, such as UML Activity Diagrams [Eshuis and Wieringa, 2004]. On the other hand, inter-model consistency deals with relations between several models that may represent different views on the same system or application. For instance, inter-model consistency of UML Class Diagrams and UML State Machines requires that all the operations used in the state machines are defined in the corresponding classes in the class diagrams [Van Der Straeten et al., 2003].

Both types of consistency are strongly related to model semantics. Consistency for a given set of models can be defined by first establishing required consistency proper-

ties on the semantic level and then defining conditions in terms of syntactic model elements directly, such that the satisfaction of these conditions guarantees semantic consistency [Küster, 2004]. For example, existence of an instance of a given class diagram is a semantic consistency property that requires an understanding of instantiation semantics of UML Class Diagrams. Conditions ensuring this property can be defined in terms of classes, associations and other syntactic model elements, which makes it possible to directly evaluate these conditions over the elements of a given class diagram.

Since intra-model consistency can be seen as a type of well-formedness or correctness of a particular model, establishing intra-model consistency is a prerequisite for defining inter-model consistency. Therefore, we begin this chapter by addressing the correctness of an object state specification in a process model as an intra-model consistency issue of process models. We first provide an introduction to the existing work on correctness of control and data flow in process models, and then proceed to defining the correctness of a process model with a state specification. Since object life cycle models are far less complex than process models, we do not identify any intra-model consistency issues for them. In the second part of the chapter, we address the inter-model consistency for process and object life cycle models.

4.2 Intra-Model Consistency of a Process Model

Definition and evaluation of control-flow correctness in process models has already been extensively studied in the existing literature. Some work has also been done around the correctness of data flow. We therefore begin by providing an overview of the existing work on these topics.

4.2.1 Existing Notions of Control-Flow and Data-Flow Correctness

Control-flow correctness of process models is generally defined in terms of *soundness*, originally introduced by van der Aalst [van der Aalst, 1997]. Assuming a control-flow semantics based on tokens, like the one defined in Chapter 3, soundness requires that the terminal execution state is reachable from every other execution state of a process model, and when the terminal execution state is reached, there are no tokens anywhere else in the process model. In the original definition, soundness also requires that for every action in the process model, there exists at least one execution sequence containing this action. However, some later uses of the soundness notion (e.g. [Vanhatalo et al., 2007]), only focus on the first two requirements that ensure proper termination of the execution of a process model.

In more recent work, soundness has also been explained in terms of the absence of structural process model properties that lead to *deadlocks* and *lack of synchronization* [Sadiq and Orlowska, 2000, Vanhatalo et al., 2007]. A deadlock occurs in a process model when alternative branches introduced by a decision are at some point combined with a join, while a lack of synchronization occurs when forked concurrent paths are later combined with a merge, which can result in unintentional multiple activations of succeeding nodes.

Since the semantics of data flow is not yet as well-established as that of control flow, less work has been done on defining and checking data-flow correctness. In [Sadiq et al., 2004], Sadiq et al give a high-level overview of different data-flow correctness problems that can occur in a process model, comprising missing, misdirected, lost, inconsistent, redundant, mismatched and insufficient data. The authors point out

that *how* such problems are tackled depends on the way data flow is represented in a process modeling language and its semantics. In this dissertation, we focus on repository and routed data flow, as defined in Chapter 3.

As introduced in [Sadiq et al., 2004], *missing data* essentially refers to a deadlock induced by data flow, which occurs if an action in a process model requires an object that is not created by any of the action's predecessors. While this problem can occur in process models with repository data flow, routed data flow does not suffer from it, since typed edges always connect an object provider to an object receiver. *Misdirected data* refers to a situation where an action cannot get an object from its intended object provider due to the control flow in the process model. Variations of this problem can occur in both, repository and routed data flow. *Lost data* describes a situation where updates to the same object are performed by concurrent actions, in which case one of the updates may be "lost"; while *inconsistent data* generalizes this situation to concurrent updates external to the process model. On the other hand, *redundant data* characterizes situations where a created object does not subsequently serve as an input to any action. These three problems can only occur in repository data flow, where multiple actions can update and read the same object in a repository. Finally, *mismatched data* and *insufficient data* respectively take into account the internal object structure and internal action semantics, which are not included in our definitions of repository and routed data flow.

Out of all the above-mentioned data-flow correctness problems, only missing data has so far been addressed for a particular type of process models and data flow. Mendling et al [Mendling et al., 2008] extend the notion of control-flow soundness to EPCs [Keller et al., 1992] with object flows, thereby targeting to identify missing data. Stoerrle's formalization of UML Activity Diagrams with data flow [Stoerrle, 2005] uses colored Petri nets (CPN) [Jensen, 1995] and facilitates the verification of process models with data flow against proper termination, which implicitly addresses the missing data problem.

Our goal is to establish how the extension of control and data flow with object state specifications affects the correctness of a process model. In the following, we assume process models to have sound control flow and not to contain missing data, as these properties can be ensured with existing approaches. Using the semantics defined in the previous chapter, we define *control-flow soundness* based on the definition of van der Aalst [van der Aalst, 1997] and missing data based on the informal description by Sadiq et al [Sadiq et al., 2004].

Definition 42 (Control-flow soundness)**.** *A workflow graph $G = (N, E)$ with the initial execution state w_i and the terminal execution state w_t is* control-flow sound *if and only if the following conditions hold:*

- *(proper termination) w_t is reachable from every execution state w that is reachable from w_i, and w_t is the only execution state reachable from w_i with at least one token on the incoming edge of the stop node;*
- *(no dead actions) each node $n \in N$ is activated in some execution state reachable from w_i.*

According to the above definition, control-flow soundness of a process model with data flow is determined by using the pure control-flow semantics given in Definition 3 of Chapter 3.

Since missing data can only occur in process models with repository data flow, we address this type of data flow specifically in the following definition.

Definition 43 (Missing data)**.** *A workflow graph $G = (N, E)$ with repository data flow using a set of object types T be given. G has* no missing data *if and only if given any execution state (w, O, s) of G, for all actions $a \in N$ that are control-flow activated in (w, O, s), there exists an execution state (w', O', s') of G reachable from (w, O, s) where a is control-flow activated and for all object types $t \in datain(a)$, there exists an object $o \in O'$ where $type(o) = t$.*

In the following, we describe the type of additional problems that occur when an object state specification is added to a control-flow sound process model that has no missing data and define the correctness of a state specification.

4.2.2 Correctness of a State Specification

An object state specification predominantly defines a special type of pre- and post-conditions for actions in a given process model. According to the data-flow semantics defined in the previous chapter, an action waits until its pre-conditions are satisfied to begin execution, in other words it waits until certain objects reach some of the states accepted by the action. Since we abstract from the internal details of actions, we assume that post-conditions are always satisfied after an action is executed.

As illustrated with two simple examples in Figure 4.1(a) and (b), assigning accepted and produced states of actions in a particular way can lead to a situation where it is impossible for the pre-conditions of an action to ever be satisfied. In these two examples, the $Evaluate\ Claim$ action waits forever for the $Claim$ object to be in state $Opened$. This demonstrates that state specifications of actions introduce a new source of deadlocks into a process model that is control-flow sound and has no missing data.

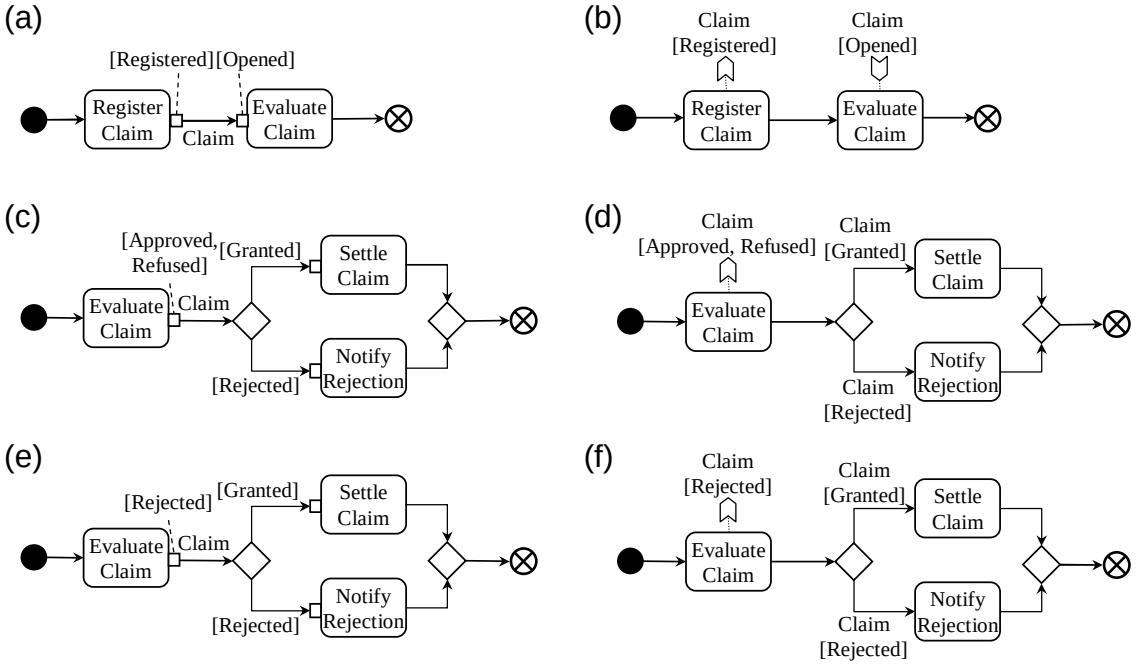


Figure 4.1: Examples of state specifications that lead to deadlocks and dead edges

Adding edge conditions to process models can also lead to deadlocks, as illustrated in the examples in Figure 4.1(c) and (d). According to the semantics defined in Chapter 3, a decision can only execute if there is a token on its incoming edge and all the edge conditions are satisfied for at least one of its outgoing edges. In these two examples,

the $Claim$ object is either in state $Approved$ or $Refused$ after the $Evaluate\ Claim$ action executes, which means that none of the outgoing edges of the decision have their edge conditions satisfied.

Even if edge conditions do not lead to a deadlock, they can result in edges that are never activated, which we refer to as *dead edges*. This is illustrated with two examples in Figure 4.1(e) and (f). In both examples, the $Evaluate\ Claim$ action always sets the state of $Claim$ to $Rejected$. As a result, the outgoing edge of the decision that has the edge condition $[Granted]$ is never activated and $Settle\ Claim$ can never be executed.

A state specification only influences the conditions under which actions and decisions can be executed and not other nodes in a process model. Therefore, the three types of situations described above represent the only types of problems that can be introduced by adding a state specification into a control-flow sound process model with no missing data. We define correctness of a state specification in a process model to avoid such situations. In the following, we define the concept of a correct state specification on the semantic level, i.e. in terms of execution states of a process model.

Definition 44 (Correct state specification)**.** *Let a workflow graph $G = (N, E)$ with data flow using a set of object types T and a state specification be given. We say that G has a* correct state specification *if the following conditions hold:*

C44.1 (state acceptance) given any execution state (w, O, s) of G, for all actions $a \in N$ that are control-flow activated in (w, O, s), there exists an execution state (w', O', s') of G reachable from (w, O, s) where a is control-flow activated and

- *if G has repository data flow, $s'(o) \in acpt(a, type(o))$ for all objects $o \in O'$ where $type(o) \in datain(a)$;*
- *if G has routed data flow, for each edge $e \in in(a)$, $s'(o_e) \in acpt(a, type(o_e))$ for all objects $o_e \in O'$;*

C44.2: (decision satisfaction) given any execution state (w, O, s) of G, for all decisions $d \in N$ that are control-flow activated in (w, O, s), there exists an execution state (w', O', s') of G reachable from (w, O, s) where d is control-flow activated and

- *if G has repository data flow, there exists an edge $e' \in out(d)$ such that $s'(o) \in cond(e', type(o))$ for all objects $o \in O'$ whenever $cond(e', type(o))$ is defined;*
- *if G has routed data flow, there exists an edge $e' \in out(d)$ such that $s'(o_e) \in cond(e', type(o_{e'}))$ for all objects $o_e \in O'$ where $e \in in(d)$ whenever $cond(e', type(o_{e'}))$ is defined.*

C44.3: (no dead edges) for each edge $e' \in E$ where $e' \in out(d)$ for some decision $d \in N$, there exists an execution state (w, O, s) where d is control-flow activated and

- *if G has repository data flow, $s(o) \in cond(e', type(o))$ for all objects $o \in O$ whenever $cond(e', type(o))$ is defined;*
- *if G has routed data flow, $s(o_e) \in cond(e', type(o_{e'}))$ for all objects $o_e \in O$ where $e \in in(d)$ whenever $cond(e', type(o_{e'}))$ is defined.*

Definition 44 formally expresses that a state specification is correct only if every action and decision that become control-flow activated can eventually execute and there are no

dead edges. According to this definition, we can conclude that none of the process models shown in Figure 4.1 have a correct state specification, by considering their possible execution sequences. However, in order to evaluate the correctness of a state specification statically, we need to define correctness conditions that can be checked on the syntactic level. We proceed to defining such syntactic correctness conditions in the following section.

4.2.3 Syntactic Correctness Conditions

We first use several examples to demonstrate how the structure of the process model and the type of data flow influence the correctness of a state specification. Figure 4.2 shows several examples with different types of data flow, where each example represents a part of a process model, as opposed to complete process models shown in Figure 4.1. To determine whether the state specification is correct in the process model extract shown in (a), we need to determine whether the $Claim$ objects always reach the $Evaluate\ Claim$ action in the $Opened$ state, as this is the only state that this action can accept. Since we always interpret routed data flow as pass-by-value, it is clear that the routed $Claim$ object cannot change its state between the execution of the $Register\ Claim$ and $Evaluate\ Claim$ actions. Since the $Register\ Claim$ action always satisfies its state post-condition and produces a $Claim$ object in state $Registered$, $Claim$ objects never reach the $Evaluate\ Claim$ action in state $Opened$. Therefore, state acceptance does not hold and this object state specification is not correct.

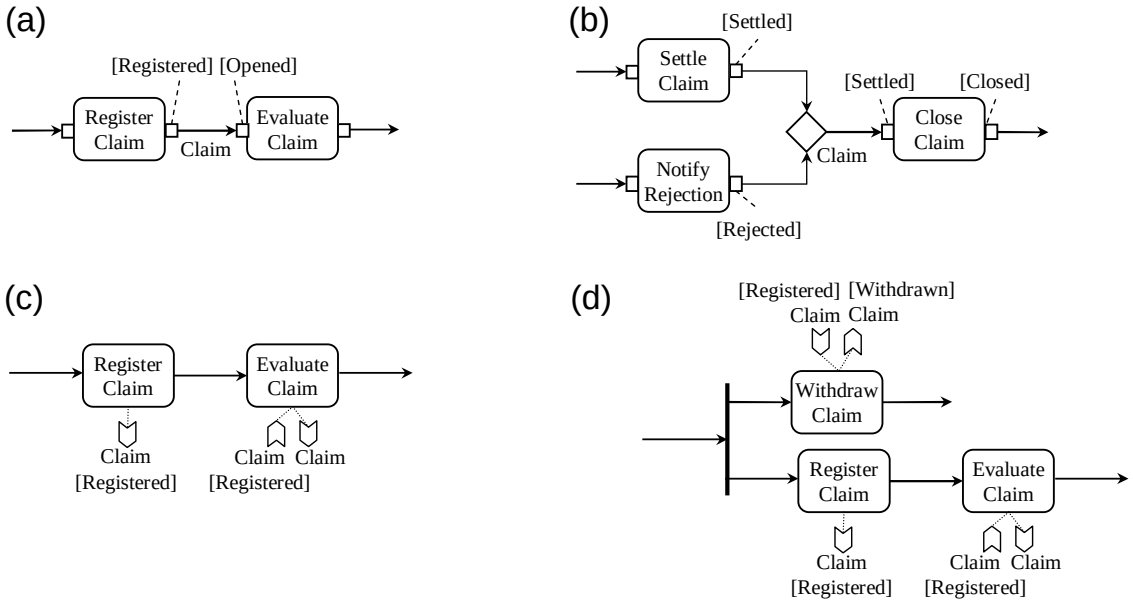


Figure 4.2: Complexities in evaluating correctness of a state specifications

In the example in Figure 4.2(b), the state of the $Claim$ objects before the execution of $Close\ Claim$ is not deterministic, since either the $Settle\ Claim$ action or the $Notify\ Rejection$ action is executed before the merge, producing a $Claim$ object in either state $Settled$ or $Rejected$. To ensure correctness, the $Close\ Claim$ action should accept $Claim$ objects in all the states that such objects can be in at that point in time, i.e. $Settled$ and $Rejected$. Since $Rejected$ is not an accepted state of $Close\ Claim$, this object state specification is not correct either.

The third example shown in Figure 4.2(c) uses repository data flow. Since repository data flow represents object passing by reference, we can no longer assume that a $Claim$ object does not change the state between the execution of the $Register\ Claim$ action and the $Evaluate\ Claim$ action. Figure 4.2(d) shows one extension of this process model extract, where the $Withdraw\ Claim$ action can be executed after the $Register\ Claim$ action and before the $Evaluate\ Claim$ action, changing the state of the $Claim$ object from $Registered$ to $Withdrawn$. Therefore, the correctness of the process model extract in (c) cannot be determined without the complete process model, and the extract shown in (d) represents an state specification that is not correct. This illustrates that in general, evaluating correctness of a state specification in a process model with repository data flow is much more complex than for routed data flow.

In the following, we define syntactic correctness conditions over elements of a process model. We show that they are necessary and sufficient for determining the correctness of a state specification according to Definition 44 for the class of process models that satisfy the following properties:

- P1: control-flow soundness;
- P2: no missing data;
- P3: one of the following:
  - P3-1: have repository data flow and no forks;
  - P3-2: have routed data flow.

To simplify the definition of the syntactic correctness conditions, we first introduce the concepts of an *object provider* and *effective input* and *output states*. On the semantic level, a node n_1 is considered to be an object provider of another node n_2 with respect to a particular object type t, if there exists an execution sequence where n_1 updates the state of an object of type t and subsequently that object is not updated until it is taken as an input by node n_2. In the following, we capture this concept precisely in syntactic terms.

Definition 45 (Object provider)**.** *Let a workflow graph $G = (N, E)$ with either repository or routed data flow using a set of object types T, nodes $n_1, n_2 \in N$ and an object type $t \in T$ be given such that $t \in dataout(n_1)$ and $t \in datain(n_2)$. Node n_1 is an* object provider *of n_2 with respect to t, written $n_1 \lhd_t n_2$, if and only if the following conditions hold:*

- *n_1 is either an action or a decision, and n_2 is either an action or a decision;*
- *there is a path $p = n_1, ..., n_2$ in G such that p does not contain any other action or decision $n_3 \in N$ where $t \in dataout(n_3)$;*
- *if G has routed data flow, $type(e) = t$ for all edges e on path p.*

As defined above, object providers can be determined by examining data inputs and outputs of actions and decisions, and the paths between these nodes in a given process model. The object provider relation can be depicted as an *object provider graph*, which is a directed graph where each node represents an action or a decision and each edge represents an object provider pair. Figure 4.3(a) shows our example claims handling process model with routed data flow, with abbreviated action, object type and state names. In this process model, the following object provider pairs exist with respect to object type

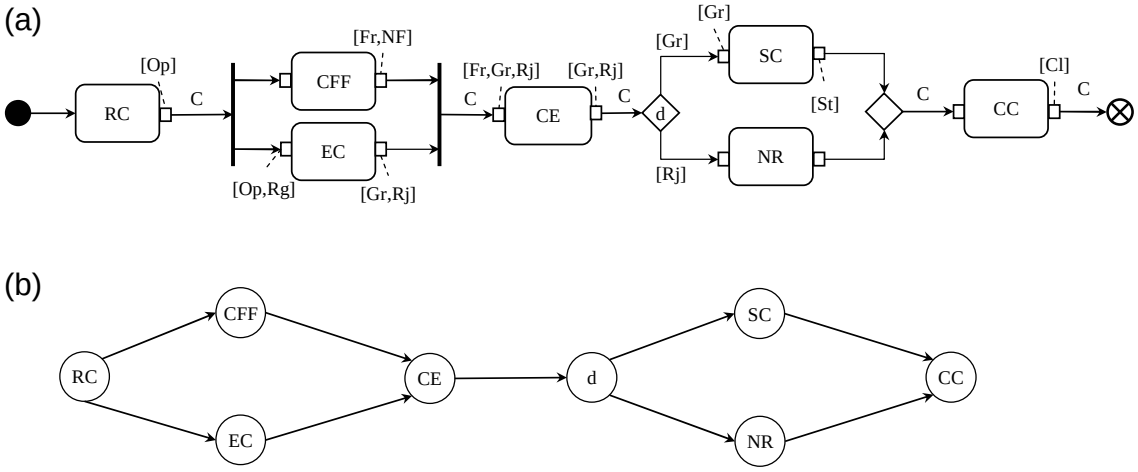


Figure 4.3: Process model and object provider graph

C ($Claim$): $RC \lhd_C CFF$, $RC \lhd_C EC$, $CFF \lhd_C CE$, $EC \lhd_C CE$, $CE \lhd_C d$, $d \lhd_C SC$, $d \lhd_C NR$, $SC \lhd_C CC$, $NR \lhd_C CC$. These are shown in an object provider graph in Figure 4.3(b).

Effective input and output states are next defined for actions and decisions, to capture all the possible states that objects of a particular type can be in before and after the execution of each of these nodes.

Definition 46 (Effective input and output states)**.** *Let a workflow graph $G = (N, E)$ with either repository or routed data flow using a set of object types T and a state specification be given. Furthermore, let a node $n \in N$ that is either an action or a decision, an object type $t \in T$, and a set N_p comprising all object providers of n with respect to t be given. We define two functions: $effin : N \times T \to \mathcal{P}(S_T)$ to map a node $n \in N$ and an object type $t \in T$ to a set of* effective input states *of n for t, and $effout : N \times N \times T \to \mathcal{P}(S_T)$ to map nodes $n, n_s \in N$ and an object type $t \in T$ to a set of* effective output states *of n for t with respect to n_s. These functions are defined as follows:*

$$effin(n,t) = \bigcup_{n_p \in N_p} effout(n_p, n, t)$$

$$effout(n, n_s, t) = \begin{cases} prod(n,t) & \text{if } n \text{ is an action and } t \notin datain(n) \\ \bigcup_{s \in effin(n,t)} dep(n,t,s) & \text{if } n \text{ is an action and } t \in datain(n) \\ \bigcup_{e_s \in E_s} cond(e_s, t), \text{ where } E_s \subseteq out(n) & \text{if } n \text{ is a decision} \\ \text{and each edge in } E_s \text{ lies on a path to } n_s & \end{cases}$$

Effective input states of an action or a decision for a particular object type are defined as a union of the effective output states of all its object providers with respect to this type, leading to an empty set if there are no object providers. In turn, effective output states of an action for a given object type are the states that can be produced by the action for this type, taking into account the dependencies on the effective input states. Effective output states of a decision for a given object type are defined in connection with another node, as the union of all states in the edge conditions of the outgoing edges of the decision that lie on a path to the other node.

Figure 4.4 shows effective input and output states using the object provider graph derived for object type C ($Claim$) from the process model shown in Figure 4.3(a). For

each node, accepted and produced states for C are respectively indicated as $[]_{acpt}$ and $[]_{prod}$, and effective input and output states as $\langle\rangle_{in}$ and $\langle\rangle_{out}$. Defaults assigned to accepted and produces states of nodes where these were unspecified in the original process model in Figure 4.3(a) are shown in Figure 4.4. For example, since there are no accepted states specified for the action CFF in Figure 4.3(a), the set of accepted states for this action is assigned to the complete set of states for object type C, i.e. $\{Op, Fr, NF, Rg, Gr, Rj, St, Cl\}$. All dependency state sets are also assigned by default. These are assigned such that any state in the accepted states can lead to any of the produced states for all nodes, except for action NR ($Notify\ Rejection$). The 1:1 dep state sets annotation for action NR indicates that each accepted state contains only itself in its dependency state set for this action. This is a result of the default state specification assignment (cf. Definition 15), since NR has no produced states specified in the original process model.

It can be seen in Figure 4.4 that Op is the only effective output state of RC for C. Since C is not a data input of action RC, its effective output states are determined directly from its produced states. Op is also the only effective input state of CFF and EC for C. Fr and NF are the effective output states of the action CFF, since the dependency state set of Op for this action comprises all the produced states, i.e. Fr and NF. Gr is the only effective output state of the decision d for C with respect to SC, while Rj is the only effective output state of the decision d for C with respect to NR.

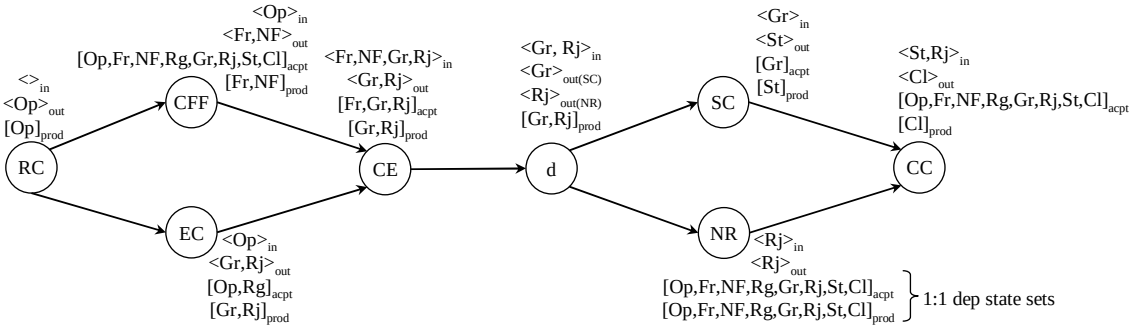


Figure 4.4: Effective states in object provider graph

The concept of effective states is essential not only for the upcoming definition of the syntactic correctness conditions, but also for expressing consistency of and transformations between process and object life cycle models later in the dissertation. For the moment, it is important to understand the definition of effective states and not necessarily how effective states are computed for a given process model. In Section 4.4, we discuss the computation of effective states.

In the following, we define the syntactic correctness conditions to check that a plausible relation exists between effective input states and accepted states of actions and edge conditions of decisions in a given process model.

Definition 47 (Syntactic correctness conditions)**.** Syntactic correctness conditions *for a given workflow graph $G = (N, E)$ with either repository or routed data flow using a set of object types T and a state specification are defined as follows:*

C47.1: for each action $a \in N$ and object type $t \in datain(a)$, $effin(a, t) \subseteq acpt(a, t)$;

C47.2: for each decision $d \in N$ and object type $t \in datain(d)$, $effin(d, t) \subseteq \bigcup_{e \in out(n)} cond(e, t)$.

C47.3: for each edge $e \in E$ where $e \in out(d)$ for some decision $d \in N$ and for each object type $t \in datain(d)$, $cond(e,t) \cap effin(d,t) \neq \emptyset$.

In the example shown in Figure 4.4, the syntactic correctness condition C47.2 is satisfied, while C47.1 is violated, because state NF ($NoFraud$) is an effective input state for the action CE ($Confirm\ Evaluation$), but it is not in the set of its accepted states. This implies that the state specification in this process model is not correct, meaning that a deadlock may occur during the execution of this process model due to its state specification. In the following, we show that the syntactic correctness conditions are necessary and sufficient with regards to the definition of a correct state specification for our selected class of process models.

4.2.4 Necessity and Sufficiency of Syntactic Correctness Conditions

In the following, we stipulate and prove the relation between the syntactic correctness conditions (Definition 47) and the correctness of a state specification (Definition 44). We first define two useful lemmas about the relation of execution sequences of a workflow graph and paths in a workflow graph.

Lemma 1. *Let a workflow graph $G = (N, E)$ with repository data flow using a set of object types T and a state specification be given such that N contains no forks (P3-1). Let (w_m, O_m, s_m) be an execution state of G, in which a node $n_m \in N$ that is an action or a decision is control-flow activated and there is an object $o \in O_m$ of type t such that $t \in datain(n_m)$. Then, there exists an execution sequence $q = (w_0, O_0, s_0) \overset{n_0}{\rightarrow} (w_1, O_1, s_1)...(w_{m-1}, O_{m-1}, s_{m-1}) \overset{n_{m-1}}{\rightarrow} (w_m, O_m, s_m)$ such that $n_0 \lhd_t n_m$ and $s_i(o) = s_m(o)$ for all $0 < i < m$.*

Proof. Since the object set is empty in the initial execution state, some action a must have been executed to create o before the execution state (w_m, O_m, s_m) was reached. The state of o has not changed either after the execution of a until (w_m, O_m, s_m) (case 1) or after the execution of another action a' executed after a but before (w_m, O_m, s_m) (case 2). Since G has no forks, for every execution sequence $(w_x, O_x, s_x) \overset{n_x}{\rightarrow} (w_{x+1}, O_{x+1}, s_{x+1})...(w_{z-1}, O_{z-1}, s_{z-1}) \overset{n_{z-1}}{\rightarrow} (w_z, O_z, s_z)$ there is a path $n_x, ..., n_{z-1}$ in G. Hence, there must be a path p to n_m that begins either with a (case 1) or a' (case 2). In either way, there must be a node n_0 that is an object provider of n_m with respect to t on path p. This implies that there is an execution sequence q with the properties stated in Lemma 1. □

Lemma 2. *Let a workflow graph $G = (N, E)$ with routed data flow using a set of object types T and a state specification be given (P3-2). Let (w_m, O_m, s_m) be an execution state of G, in which a node $n_m \in N$ that is an action or a decision is control-flow activated. Then, for each object $o_e \in O_m$ of type t where $e \in in(n_m)$, there exists an execution sequence $q = (w_0, O_0, s_0) \overset{n_0}{\rightarrow} (w_1, O_1, s_1)...(w_{m-1}, O_{m-1}, s_{m-1}) \overset{n_{m-1}}{\rightarrow} (w_m, O_m, s_m)$ such that:*

- *L2.1: $n_0 \lhd_t n_m$ with $p = n_0, ..., n_m$ being a path that does not contain any other action or decision $n \in N$ where $t \in dataout(n)$ (cf. Definition 45);*
- *L2.2: and for all objects $o_{e'}$ labeled with an edge e' on path p where $e' \neq e$, $o_{e'} \in O_i \Rightarrow s_i(o_{e'}) = s_m(o_e)$ for all $0 < i < m$.*

Proof. Let an object $o_e \in O_m$ where $e \in in(n_m)$ be given. Since n_m is control-flow activated in (w_m, O_m, s_m) and has an incoming edge of type t, at least one object provider of n_m with respect to t must have been executed before (w_m, O_m, s_m). Hence, there is an execution sequence q as defined in the lemma where L2.1 holds. To show L2.2, there are two cases to consider:

Case 1 (without join): On the one hand, node n_m can be either a decision or an action that is not preceded by a join. In this case, for each executed object provider n_0 of n_m with respect to t, no path $p = n_0, ..., n_m$ that satisfies the condition given in L2.1 can contain joins, since each join with typed edges is directly followed by a reconciliation action in routed data flow. Hence, such a path p can only contain merges and forks. An execution of a merge with typed edges relabels one object without changing its state (cf. Definition 35). An execution of a fork with typed edges replaces one object by several objects and sets the state of the replacement objects to the state of their prior object (cf. Definition 31). Therefore, object o_e must be a result of zero or more relabelings and replacements of exactly one object o_{e_0} created or updated by some object provider n_0 of n_m with respect to t. Since merges and forks do not change state of objects, L2.2 must hold with respect to a path from n_0 to n_m that satisfies the condition given in L2.1.

Case 2 (with join): On the other hand, node n_m can be an action that is preceded by a join. In this case, object o_e must be a relabeling of an object o_{e_j} where e_j is an incoming edge of the join n_j that precedes n_m (cf. Definition 33). There must be an executed object provider n_0 of n_m with respect to t such that e_j lies on a path $p = n_0, ..., n_m$ that satisfies the condition given in L2.1. Apart from n_j, p can only contain merges and forks. Therefore, object o_{e_j} must be a result of zero or more relabelings and replacements of exactly one object o_{e_0} created or updated by n_0. Since merges, forks and joins do not change the state of objects, L2.2 must hold with respect to p. □

Lemma 3. *Let a workflow graph $G = (N, E)$ with repository data flow using a set of object types T be given such that N contains no forks (P3.1), and G satisfies properties P1, P2 and has a correct state specification. Furthermore, let an object type $t \in T$ and nodes $n_0, n_m \in N$ where $n_0 \lhd_t n_m$ be given such that $p = n_0, ..., n_m$ is a path that does not contain any other action or decision $n \in N$ where $t \in dataout(n)$ (cf. Definition 45). Then, there exists an execution sequence $q = (w_0, O_0, s_0) \overset{n_0}{\rightarrow} (w_1, O_1, s_1)...(w_m, O_m, s_m) \overset{n_m}{\rightarrow} (w_{m+1}, O_{m+1}, s_{m+1})$ such that for some object $o \in O_m$ of type t, $s_m(o) = s_i(o)$ for all $1 \leq i < m$.*

Proof. Since G satisfies properties P1, P2, P3.1 and has a correct state specification, there must be an execution sequence $q = (w_0, O_0, s_0) \overset{n_0}{\rightarrow} (w_1, O_1, s_1)...(w_m, O_m, s_m) \overset{n_m}{\rightarrow} (w_{m+1}, O_{m+1}, s_{m+1})$ such that q only comprises the execution of nodes that lie on path p. We know that p contains no action a where $t \in dataout(a)$, $a \neq n_0$ and $a \neq n_m$. Hence, no such action is executed between (w_1, O_1, s_1) and $(w_{m-1}, O_{m-1}, s_{m-1})$. Since only actions with data outputs of a particular type change the state of objects of that type, it means that the state of all objects of type t remains the same in all execution states (w_i, O_i, s_i) where $1 \leq i \leq m$. Therefore, q satisfies the properties stated in Lemma 3. □

Lemma 4. *Let a workflow graph $G = (N, E)$ with routed data flow using a set of object types T be given (P3.1) such that G satisfies properties P1, P2 and has a correct state specification. Furthermore, let an object type $t \in T$, nodes $n_0, n_m \in N$ where $n_0 \lhd_t n_m$ be given such that $p = n_0, ..., n_m$ is a path that does not contain any other action or decision $n \in N$ where $t \in dataout(n)$ (cf. Definition 45). Then, there exists an execution sequence $q = (w_0, O_0, s_0) \overset{n_0}{\rightarrow} (w_1, O_1, s_1)...(w_m, O_m, s_m) \overset{n_m}{\rightarrow} (w_{m+1}, O_{m+1}, s_{m+1})$ such that for some object $o_e \in O_m$ of type t where $e \in in(n_m)$, for all objects $o_{e'}$ labeled with an edge e' on path p where $e' \neq e$, $o_{e'} \in O_i \Rightarrow s_m(o_e) = s_i(o_{e'})$ for all $1 \leq i < m$.*

Proof. Since G satisfies properties P1, P2 and has a correct state specification, there must be an execution sequence $q = (w_0, O_0, s_0) \overset{n_0}{\rightarrow} (w_1, O_1, s_1)...(w_m, O_m, s_m) \overset{n_m}{\rightarrow} (w_{m+1}, O_{m+1}, s_{m+1})$ such that q comprises the execution of nodes that lie on path p and possibly the execution of other nodes. The execution of nodes that do not lie on p does not affect the state of the object o_{e_0} that was created or updated by n_0. The path p may contain forks, joins and merges. Since the execution of these nodes only relabels or replaces objects without changing the state of objects, the object o_{e_0} may have been relabeled or replaced but has not changed its state between (w_1, O_1, s_1) and (w_m, O_m, s_m). Therefore, q satisfies the properties stated in Lemma 4 with object o_e being a result of zero of more relabelings and replacements of object o_{e_0}. □

Theorem 1. *Let a workflow graph $G = (N, E)$ with repository data flow using a set of object types T be given such that N contains no forks (P3.1), and G satisfies properties P1, P2 and has a correct state specification. Then, the syntactic correctness conditions defined in Definition 47 hold if and only if G has a correct state specification according to Definition 44.*

Proof. In the following we prove several condition implications to prove the theorem.

C47.1 ⇒ C44.1: Suppose that C47.1 holds. Let (w, O, s) be an execution state of G such that an action $a \in N$ is control-flow activated in (w, O, s). We use proof by contradiction to show that for all $o \in O$ where $type(o) \in datain(a)$, $s(o) \in \mathit{effin}(a, type(o))$ and therefore $s(o) \in acpt(a, type(o))$. As the original hypothesis, we suppose that C44.1 does not hold, i.e. for some object $o \in O$ of type $t \in datain(a)$, $s(o) \notin \mathit{effin}(a, t)$ (**H1**).

By definition of effective input states, H1 implies that $s(o) \notin \mathit{effout}(n, a, t)$ for all object providers n of a with respect to t (**H1'**). By Lemma 1, we know that a has object providers with respect to t, so let n_0 be an object provider of a with respect to t such that the execution sequence $q = (w_0, O_0, s_0) \overset{n_0}{\rightarrow} (w_1, O_1, s_1)...(w_{m-1}, O_{m-1}, s_{m-1}) \overset{n_{m-1}}{\rightarrow} (w_m, O_m, s_m)$ where $(w_m, O_m, s_m) = (w, O, s)$ has the properties described in Lemma 1. There are three cases to consider:

Case 1 (creation action): If n_0 is an action and $t \notin datain(n_0)$, then by definition of effective output states H1' implies that $s(o) \notin prod(n_0, t)$. Since n_0 was executed according to Definition 20, it must be that $s_1(o) \in prod(n_0, t)$. We know that the state of o has not changed between (w_1, O_1, s_1) and (w, O, s), so it must be that $s(o) \in prod(n_0, t)$, which is a contradiction.

Case 2 (update action): If n_0 is an action and $t \in datain(n_0)$, then by definition of effective output states H1' implies that $s(o) \notin \bigcup_{s' \in \mathit{effin}(n_0,t)} dep(n_0, t, s')$ (**H1"**). Since n_0 was executed according to Definition 20, it must be that $s(o) \in dep(n_0, t, s_0(o))$. Then, H1" further implies that $s_0(o) \notin \mathit{effin}(n_0, t)$. We can repeat the reasoning from our original hypothesis H1 substituting a by n_0. This case cannot occur indefinitely, because eventually the node executed after the initial execution state will be reached. If this occurs, it means o was not created by any of the actions executed before (w, O, s). This implies a non-empty object set in the initial execution state, which is a contradiction.

Case 3 (decision): Otherwise n_0 must be a decision, in which case by definition of effective output states H1' implies that $s(o) \notin cond(e, t)$ for each edge $e \in out(n_0)$ that lies on a path from n_0 to a. Since n_0 was executed according to Definition 23, it must be that $s_0(o) \in cond(e_0, t)$ for some edge $e_0 \in out(n_0)$ that received a token as a result of the execution of n_0. We know that there is a path from the target of e_0 to a, hence e_0 must lie on a path from n_0 to a. Therefore, we have a contradiction.

(C44.1 ∧ C44.2 ∧ C44.3) ⇒ C47.1: Suppose that C44.1, C44.2 and C44.3 hold. We use proof by contradiction to show that for all actions $a \in N$ and object types $t \in datain(a)$, $\mathit{effin}(a, t) \subseteq acpt(a, t)$. Let an action a and an object type t be given and let (w, O, s) be

an execution state, in which a is activated (such an execution state exists since G satisfies P1, P2 and C44.1, C44.2 and C44.3 hold). As the original hypothesis, we suppose that there exists a state $s_a \in S_t$ such that $s_a \in \mathit{effin}(a,t)$ and $s(o) \neq s_a$ for all objects $o \in O$ of type t (**H2**).

By definition of effective input states, H implies that $s_a \in \mathit{effout}(n_0,a,t)$ for some object provider n_0 of a with respect to t (**H2'**). There are three cases to consider:

Case 1 (creation action): If n_0 is an action and $t \notin \mathit{datain}(n_0)$, then by definition of effective output states H2' implies that $s_a \in \mathit{prod}(n_0,t)$. This means that it is possible that n_0 creates an object of type t in state s_a. By Lemma 3, we know that there must be an execution sequence $q = (w_0,O_0,s_0) \overset{n_0}{\rightarrow} (w_1,O_1,s_1)...(w_{m-1},O_{m-1},s_{m-1}) \overset{n_{m-1}}{\rightarrow} (w_m,O_m,s_m)$ where $(w_m,O_m,s_m) = (w,O,s)$ such that $s(o) = s_a$ for some object $o \in O$ of type t. This is a contradiction.

Case 2 (update action): If n_0 is an action and $t \in \mathit{datain}(n_0)$, then by definition of effective output states H2' implies that $s_a \in \bigcup_{s'_a \in \mathit{effin}(n_0,t)} \mathit{dep}(n_0,t,s'_a)$. We can repeat the reasoning from our original hypothesis H2 substituting s_a by $s'_a \in \mathit{effin}(n_0,t)$. This case cannot occur indefinitely provided that the same action is only examined once, because eventually an action with no object providers will be reached and Case 1 applied.

Case 3 (decision): Otherwise n_0 must be a decision, in which case by the definition of effective output states H2' implies that $s_a \in \mathit{cond}(e_0,t)$ for some $e_0 \in \mathit{out}(n_)$ that lies on a path from n_0 to a. This means that it is possible that the state of an object of type t is s_a after the execution of n_0. By Lemma 3, we know that there must be an execution sequence $q = (w_0,O_0,s_0) \overset{n_0}{\rightarrow} (w_1,O_1,s_1)...(w_{m-1},O_{m-1},s_{m-1}) \overset{n_{m-1}}{\rightarrow} (w_m,O_m,s_m)$ where $(w_m,O_m,s_m) = (w,O,s)$ such that $s(o) = s_a$ for some object $o \in O$ of type t. This is a contradiction.

Similar type of argumentation can be used to show that C47.2 $\Rightarrow$ C44.2, C47.3 $\Rightarrow$ C44.3, (C44.1 $\wedge$ C44.2 $\wedge$ C44.3) $\Rightarrow$ C47.2 and (C44.1 $\wedge$ C44.2 $\wedge$ C44.3) $\Rightarrow$ C47.3. □

Theorem 2. *Let a workflow graph $G = (N,E)$ with routed data flow using a set of object types T be given (P3.1) such that G satisfies properties P1, P2 and has a correct state specification. Then, the syntactic correctness conditions defined in Definition 47 hold if and only if G has a correct state specification according to Definition 44.*

Theorem 2 can be proven using the same line of argumentation as for Theorem 1 applying Lemmas 2 and 4.

We have now defined conditions that can be evaluated on the syntactic elements of a process model and have shown that these conditions are sufficient and necessary to ensure correctness of a state specification in our selected class of process models with properties P1, P2 and P3 described earlier in the section. This concludes our investigation of intra-model consistency of process models. As already mentioned, we do not find the need to establish additional intra-model consistency properties for object life cycle models. In the next section, we move onto defining inter-model consistency for process and object life cycle models.

4.3 Inter-Model Consistency of Process and Object Life Cycle Models

Definition of inter-model consistency requires a more elaborate approach compared to intra-model consistency, since the semantics of the underlying models first need to be

unified. A consistency management methodology proposed by Küster [Küster, 2004] explicitly captures this semantic unification as a mapping of the different model types to a common semantic domain, as illustrated in Figure 4.5. One of the central notions in this methodology is that of a *consistency concept*, which captures a generic approach to defining consistency for a set of models. The mapping to a common semantic domain assists the identification of the semantic overlap between several given model types. Consistency is then defined in terms of *consistency conditions* that can refer to the syntactic model elements, as well as the elements in the common semantic domain. These consistency conditions can be checked on a set of concrete models of types $1, ..., n$ to determine whether they are consistent or not with regards to the consistency concept.

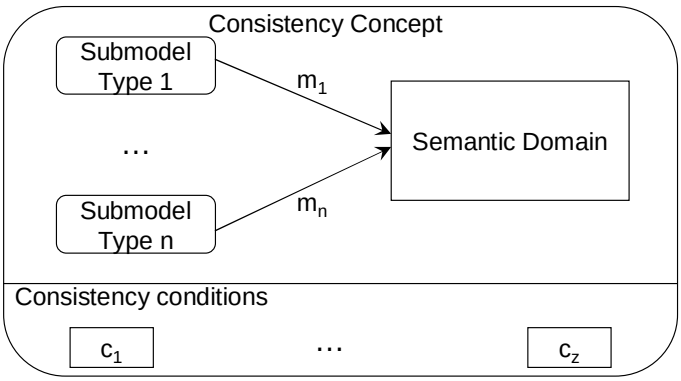


Figure 4.5: Consistency concept introduced in [Küster, 2004]

Other approaches to consistency management also emphasize the importance of model semantics and often map models to a formal language for consistency checking [Große-Rhode, 2001, Bhaduri and Venkatesh, 2002, Küster and Stehr, 2003, Van Der Straeten, 2005].

In this part of the chapter, we use the main concepts of the methodology proposed by Küster: we first establish a common semantic domain for process and object life cycle models and then define a set of consistency conditions. Similarly as for the correctness of a state specification, we show that these conditions are sufficient and necessary with regards to the semantic definition of the consistency for process and object life cycle models.

4.3.1 Establishing a Common Semantic Domain

Process models have an execution semantics according to which they create objects and update the states of these objects. On the other hand, object life cycle models are used to check conformance and coverage of object state sequences produced by the executions of a given system, whose behavior is a black box. In bringing these two types of models together, it is natural to consider a given process model as the behavioral specification of the system to be checked for conformance and coverage with a given object life cycle model. Hence, the system X used in the definition of object life cycle model semantics (cf. Section 3.3) seizes to be a black box, as illustrated in Figure 4.6.

As the common semantic domain, we therefore choose state sequences of objects. We extend the process model execution semantics with so-called *state histories*, so that state sequences can be conveniently derived from process execution sequences. This extension represents the mapping of process models to the common semantic domain. Since object life cycle conformance and coverage are already expressed in terms of conditions on state sequences, no further mapping is required.

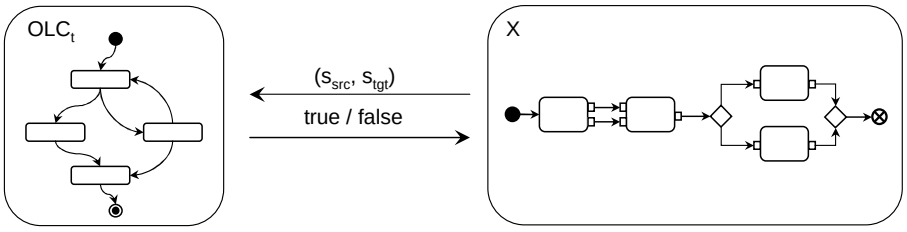


Figure 4.6: Combining process and object life cycle models

Extending Process Model Semantics with State Histories

The execution state of a workflow graph G, (w, O, s) where w is a mapping of tokens to edges, O is the set of existing objects and s is the state mapping, is extended to additionally store a state history for each object $o \in O$. A state history of an object o of type t is a set of state sequences $\{\langle s_{11}, ..., s_{1p}\rangle, ..., \langle s_{n1}, ..., s_{nq}\rangle\}$ composed of states in S_t.

Definition 48 (Execution state: extended with state history)**.** *Let a workflow graph $G = (N, E)$ with repository or routed data flow using a set of object types T and a state specification be given. An* execution state *of G is represented by (w, O, s, h), where w is the mapping of tokens to edges as defined in Definition 2, O and s are a set of objects and a mapping of objects to states as defined in Definition 17, and $h : O \rightarrow \mathcal{P}({S_T}^{seq})$ is a mapping that assigns an object to its* state history.

The $\frown$ operation is used to append a state to all state sequences in a state history, i.e. $\{\langle s_{11}, ..., s_{1p}\rangle, ..., \langle s_{n1}, ..., s_{nq}\rangle\}\frown s = \{\langle s_{11}, ..., s_{1p}, s\rangle, ..., \langle s_{n1}, ..., s_{nq}, s\rangle\}$. To avoid redundancy in state histories, we assume that a state s is only appended to a state sequence $\langle s_1, ..., s_q\rangle$ if $s \neq s_q$.

During the execution of a workflow graph with repository data flow, each object only has one state sequence in its state history. In routed data flow however, an object may have multiple state sequences in its state history, which represent the state evolution of different object copies from which this object was reconciled.

For repository data flow, we only adapt the execution rule for actions (cf. Definition 20).

Definition 49 (Execution of an action: repository data-flow, extended with state history)**.** *Let a workflow graph $G = (N, E)$ with repository data flow be given. The* execution of an action *$n \in N$ changes an execution state (w, O, s, h) to another execution state (w', O', s', h'), written $(w, O, s, h) \xrightarrow[rep,act]{n} (w', O', s', h')$, under the following pre- and post-conditions:*

- *Pre 49.1: there is at least one token on the incoming edge of n in w:*

$$\forall e \in in(n).\ w(e) > 0$$

- *Pre 49.2: objects required as inputs by n exist in O and are in accepted states:*

$$\forall t \in datain(n).\ \exists o \in O.\ type(o) = t \wedge s(o) \in acpt(n, t)$$

- *Post 49.1: in w', one token is removed from the incoming edge of n and one token is added to the outgoing edge of n:*

$$w'(e) = \begin{cases} w(e) - 1 & e \in in(n) \\ w(e) + 1 & e \in out(n) \\ w(e) & \textit{otherwise.} \end{cases}$$

- *Post 49.2: O' comprises object sets O_{old}, O_{upd} and O_{new}, where O_{old} consists of unchanged objects in O, O_{upd} consists of objects in O with their states updated and O_{new} contains new objects created by n:*

$$\begin{aligned} O' &= O_{old} \cup O_{upd} \cup O_{new} \\ O_{old} &= \{o \in O \mid type(o) \notin datain(n) \cap dataout(n) \wedge type(o) \notin dataout(n) \setminus datain(n)\} \\ &\quad \textit{and } \forall o \in O_{old}.\ s'(o) = s(o) \wedge h'(o) = h(o) \\ O_{upd} &= \{o \in O \mid type(o) \in datain(o) \cap dataout(o)\} \textit{ and} \\ &\quad \forall o \in O_{upd}.\ s'(o) \in dep(n, t, s(o)) \wedge h'(o) = h(o)^\frown s'(o) \\ O_{new} &= new_{rep}(n) \textit{ and } \forall o \in O_{new}.\ s'(o) \in prod(n, type(o)) \wedge h'(o) = \langle s'(o) \rangle \end{aligned}$$

Assuming the execution of a given a workflow graph G with repository data according to the above extension, we can obtain all the possible state sequences for an object type t by considering its state histories in the terminal execution state of G. Figure 4.7 shows an adapted claims handling process model with repository data flow. The execution sequences of this process model include x_1, x_2, x_3 and x_4, each of which begins with the initial execution state and ends with the terminal execution state. Below the process model in Figure 4.7, the state histories for object type $Claim$ in the terminal execution state of execution sequences x_1, x_2, x_3 and x_4 are shown. A union of all state histories for the object type $Claim$ comprises all the possible state sequences that can result from the executions of this process model. This example demonstrates that there may be an infinite number of such state sequences.

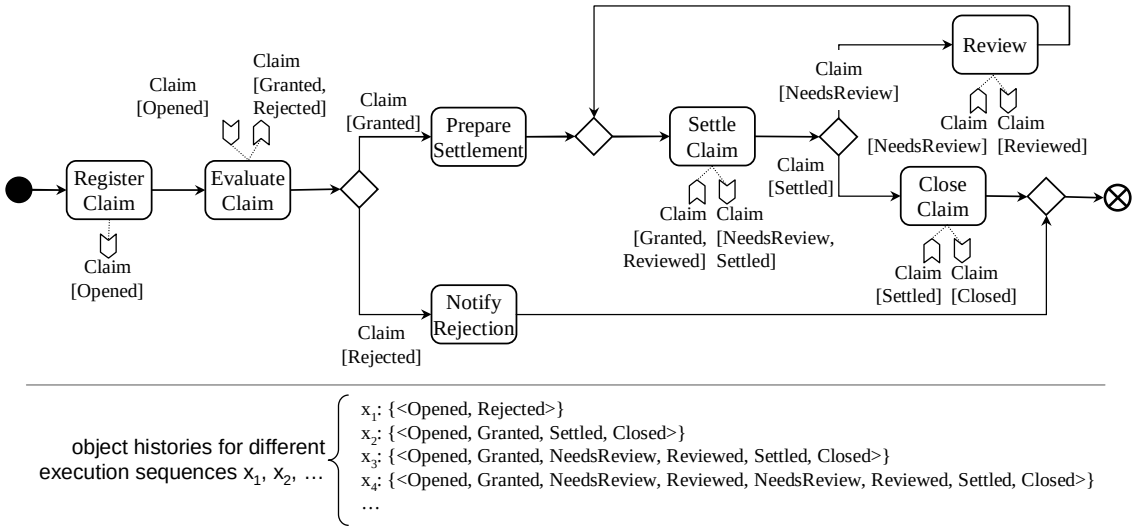


Figure 4.7: State histories in repository data flow

For routed data flow, we adapt the execution rules for actions and forks (cf. Definitions 30 and 31).

Definition 50 (Execution of an action: routed data-flow, extended with state history)**.** *Let a workflow graph $G = (N, E)$ with routed data flow be given. The* execution of an action *$n \in N$ changes an execution state (w, O, s, h) to another execution state (w', O', s', h'), written $(w, O, s, h) \xrightarrow[rout,act]{n} (w', O', s', h')$, under the following pre- and post-conditions:*

- *Pre 50.1: there is at least one token on each incoming edge of n in w:*

$$\forall e \in in(n).\ w(e) > 0$$

- *Pre 50.2: all objects in O required as inputs by n are in accepted states:*

$$\forall e \in in(n).\ \forall o \in O.\ s(o) \in acpt(n, type(o))$$

- *Post 50.1: in w', one token is removed from every incoming edge of n and one token is added to every outgoing edge of n:*

$$w'(e) = \begin{cases} w(e) - 1 & e \in in(n) \\ w(e) + 1 & e \in out(n) \\ w(e) & \textit{otherwise.} \end{cases}$$

- *Post 50.2: object set O' comprises object sets O_{old}, O_{upd} and O_{new}, where O_{old} consists of unchanged objects in O, O_{upd} comprises objects created by n to replace objects in $O \setminus O_{old}$ and O_{new} contains completely new objects created by n:*

$$\begin{aligned} O' &= O_{old} \cup O_{upd} \cup O_{new} \\ O_{old} &= \{o_e \in O \mid e \notin in(n) \vee type(e) \notin dataout(n)\} \text{ and} \\ &\quad \forall o_e \in O_{old}.\ s'(o_e) = s(o_e) \wedge h'(o_e) = h(o_e) \\ O_{upd} &= repl(n) \text{ and } \forall o_e \in O_{upd}.\ s'(o_e) \in \textstyle\bigcup_{o \in prior(n,O,o_e)} dep(n, type(e), s(o)) \wedge \\ &\quad h'(o_e) = \textstyle\bigcup_{o \in prior(n,O,o_e)} hist(o)^\frown s'(o_e) \\ O_{new} &= new_{rout}(n) \text{ and } \forall o_e \in O_{new}.\ s'(o_e) \in prod(n, type(e)) \wedge h'(o_e) = \langle s'(o_e) \rangle \end{aligned}$$

Definition 51 (Execution of a fork: routed data-flow)**.** *Let a workflow graph $G = (N, E)$ with routed data flow be given. The* execution of a fork *$n \in N$ changes an execution state (w, O, s) to another execution state (w', O', s'), written $(w, O, s) \xrightarrow[rout,fork]{n} (w', O', s')$, under the following pre- and post-conditions:*

- *Pre 51.1: there is at least one token on the incoming edge of n in w:*

$$\forall e \in in(n).\ w(e) > 0$$

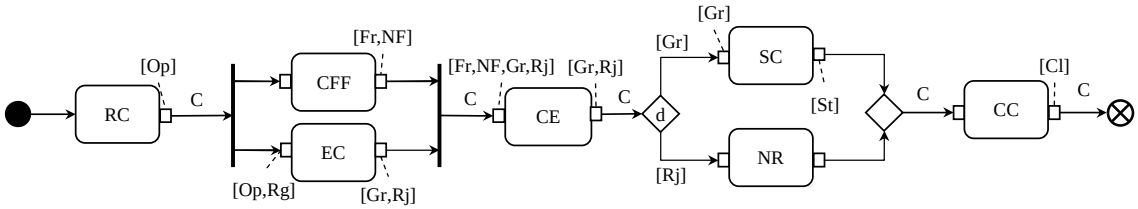
- *Post 51.1: in w', one token is removed from the incoming edge of n and one token is added to every outgoing edge of n:*

$$w'(e) = \begin{cases} w(e) - 1 & e \in in(n) \\ w(e) + 1 & e \in out(n) \\ w(e) & \textit{otherwise.} \end{cases}$$

- *Post 51.2: object set O' comprises object sets O_{old} and O_{repl}, where O_{old} contains objects in O that remain unchanged and O_{repl} comprises objects created by n to replace objects in $O \setminus O_{old}$:*

$$
\begin{aligned}
O' &= O_{old} \cup O_{repl} \\
O_{old} &= \{o_e \in O \mid e \notin in(n)\} \textit{ and } \forall o_e \in O_{old}.\ s'(o_e) = s(o_e) \wedge h'(o_e) = h(o_e) \\
O_{repl} &= repl(n) \textit{ and } \forall o_e \in O_{repl}.\ s'(o_e) \in \textstyle\bigcup_{o \in prior(n,O,o_e)} s(o) \wedge \\
&\quad h'(o_e) = \textstyle\bigcup_{o \in prior(n,O,o_e)} hist(o)^\frown s'(o_e)
\end{aligned}
$$

Figure 4.8 shows our old claims handling process model example with routed data flow. This process model has a finite number of execution sequences, which result in eight distinct state histories for object type $Claim$, as shown below the process model in Figure 4.8. As opposed to the example with repository data flow where each state history contains only one state sequence, here each state history contains two state sequences. This is a direct result of the fork creating two copies of the $Claim$ object produced by the RC action. The state of each of these objects is changed independently until they are reconciled at the CE action. When the reconciliation of objects takes place, the state sequences in their state histories are not merged, but are rather synchronously appended with new states after that point. This can be seen in the state sequences in Figure 4.8, where the last few states in the two state sequences are always the same for the same state history.



object histories for different execution sequences $x_1, \ldots, x_8$

- x_1: {<Op, Fr, Gr, St, Cl>, <Op, Gr, Gr, St, Cl>}
- x_2: {<Op, NF, Gr, St, Cl>, <Op, Gr, Gr, St, Cl>}
- x_3: {<Op, Fr, Gr, St, Cl>, <Op, Rj, Gr, St, Cl>}
- x_4: {<Op, NF, Gr, St, Cl>, <Op, Rj, Gr, St, Cl>}
- x_5: {<Op, Fr, Rf, Cl>, <Op, Gr, Rj, Cl>}
- x_6: {<Op, NF, Rj, Cl>, <Op, Gr, Rj, Cl>}
- x_7: {<Op, Fr, Rj, Cl>, <Op, Rj, Rj, Cl>}
- x_8: {<Op, NF, Rj, Cl>, <Op, Rj, Rj, Cl>}

Figure 4.8: State histories in routed data flow

In the following, we use the process model semantics extended with state histories to define process and object life cycle model consistency.

4.3.2 Consistency of Process and Object Life Cycle Models

Given a workflow graph G with repository or routed data flow and all its possible terminal execution states $(w_1, O_1, s_1, h_1), ..., (w_n, O_n, s_n, h_n)$, we define the set Q_t of state sequences for an object type t that can be generated by all possible executions of G as follows: $Q_t = \bigcup_{i=1}^{n} \{q \in h_i(o) \mid o \in O_i \wedge type(o) = t\}$.

Definition 52 (Process and object life cycle model consistency)**.** *Let a set of state sequences $Q_t = \{\langle s_{11}, ..., s_{1p}\rangle, ..., \langle s_{n1}, ..., s_{nq}\rangle\}$ for object type t generated by all possible executions of a workflow graph G be given. We say that G and a given object life cycle model $OLC_t =$*

$(S, s_\alpha, s_\omega, \Sigma, \delta)$ *are* consistent *if and only if* G *is conformant with and provides a coverage for* OLC_t *according to Definitions 40 and 41.*

Consider the above definition in the context of the process and object life cycle models shown in Figure 4.9. Object life cycle conformance requires that for every state sequence $q = \langle s_1, ..., s_r \rangle$ in Q_{Claim}, there is a transition from s_i to s_{i+1} in the object life cycle model for $Claim$ for $1 \leq i < r$. This condition clearly does not hold. For example, there is no transition from state $Granted$ to state $NeedsReview$ in the object life cycle model shown in Figure 4.9. Object life cycle coverage requires that for all transitions $s_1 \xrightarrow{e} s_2$ in the $Claim$ object life cycle model, there exists a state sequence $\langle ..., s_1, s_2, ... \rangle$ in Q_{Claim} where s_1 is directly followed by s_2. This condition does not hold either, since the transition from state $Rejected$ to state $Closed$ does not occur in any of the state sequences in Q_{Claim}. Therefore, we can conclude that the process and object life cycle model in Figure 4.9 are not consistent.

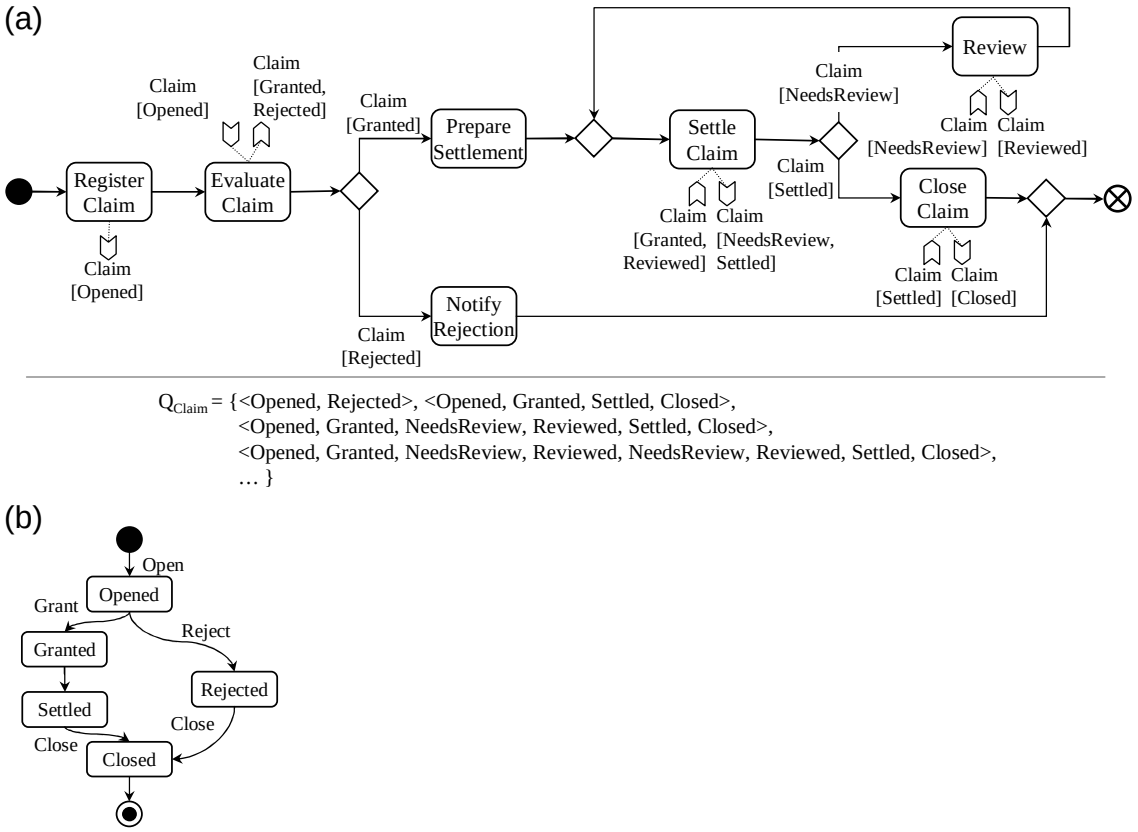


Figure 4.9: Checking object life cycle conformance and coverage

Above, we defined process and object life cycle model consistency in terms of their common semantic domain, namely state sequences. Since computing all the possible state sequences generated by a given process model is computationally expensive, in the next section we define consistency conditions that can be evaluated directly using the syntactic elements of the given models.

4.3.3 Syntactic Consistency Conditions

In the following, we define syntactic consistency conditions over elements of process and object life cycle models. We subsequently show that they are necessary and sufficient for

determining the process and object life cycle consistency according to Definition 52 for process models that satisfy properties P1, P2 and P3 defined in Section 4.2.3.

To simplify the definition of consistency conditions, we first introduce the concepts of induced transitions, first and last states. *Induced transitions* of an object type t in a given process model identify all state transitions that can occur for objects of type t during the execution of the process model.

Definition 53 (Induced transition). *Let a workflow graph $G = (N, E)$ with repository or routed data flow using a set of object types T, a state specification for G, and an object type $t \in T$ be given. The set of* induced transitions *of t in G, denoted $induced(t) \subseteq N \times S_t \times S_t$, comprises triples (a, s_1, s_2) such that the following conditions hold:*

- *a is an action;*
- *$s_1 \in effin(a, t)$;*
- *$s_2 \in dep(a, t, s_1)$;*
- *$s_1 \neq s_2$.*

As can be seen in the definition above, induced transitions are based on effective input states (defined in Section 4.2.3). Given an action a and its effective input states $effin(a, t)$, it is known that objects of type t can reach this action in any of these states for our selected class of process models. Since we assume that actions always fulfill their post-conditions, we also know that after a has completed execution the state of the object of type t is updated to one of its produced states, taking into account the dependency state sets. Naturally, if a has no outputs of type t, its produced states would be empty and thus no transitions would be induced.

Figure 4.10(a) shows the previously-introduced claims handling process model, where effective input and output states of $Claim$ are indicated for each action. Based on the effective states, induced transitions, first and last states of $Claim$ are identified. These are shown in Figure 4.10(b) using annotations in the form of $[s \rightarrow s_1, ..., s_n]$ for action a to indicate induced transitions $(a, s, s_1), ..., (a, s, s_n)$. In this example, induced transitions of $Claim$ include $(Evaluate\ Claim, Opened, Granted)$ and $(Evaluate\ Claim, Opened, Rejected)$.

First states of an object type t in a given process model are those states in which objects of type t can be created by actions in the process model. On the other hand, *last states* are the states that objects of type t can be in at the end of the process model execution.

Definition 54 (First and last states). *Let a workflow graph $G = (N, E)$ with repository or routed data flow using a set of object types T, a state specification for G, and an object type $t \in T$ be given.*

- First states *of t in G, denoted $first(t) \subseteq S_t$, comprise states s where $s \in effout(a, t)$ and $t \notin datain(a)$ for some action $a \in N$;*
- Last states *of t in G, denoted $last(t) \subseteq S_t$, comprise states s where the following conditions hold:*
  - *there exists an action or a decision $n \in N$ where $s \in effout(n, t)$;*
  - *there is a path $p = n_1, ..., n_2$ in G such that n_2 is the stop node and p does not contain any other action or decision $n_3 \in N$ where $t \in dataout(n_3)$;*
  - *if G has routed data flow, $type(e) = t$ for all edges e on path p.*

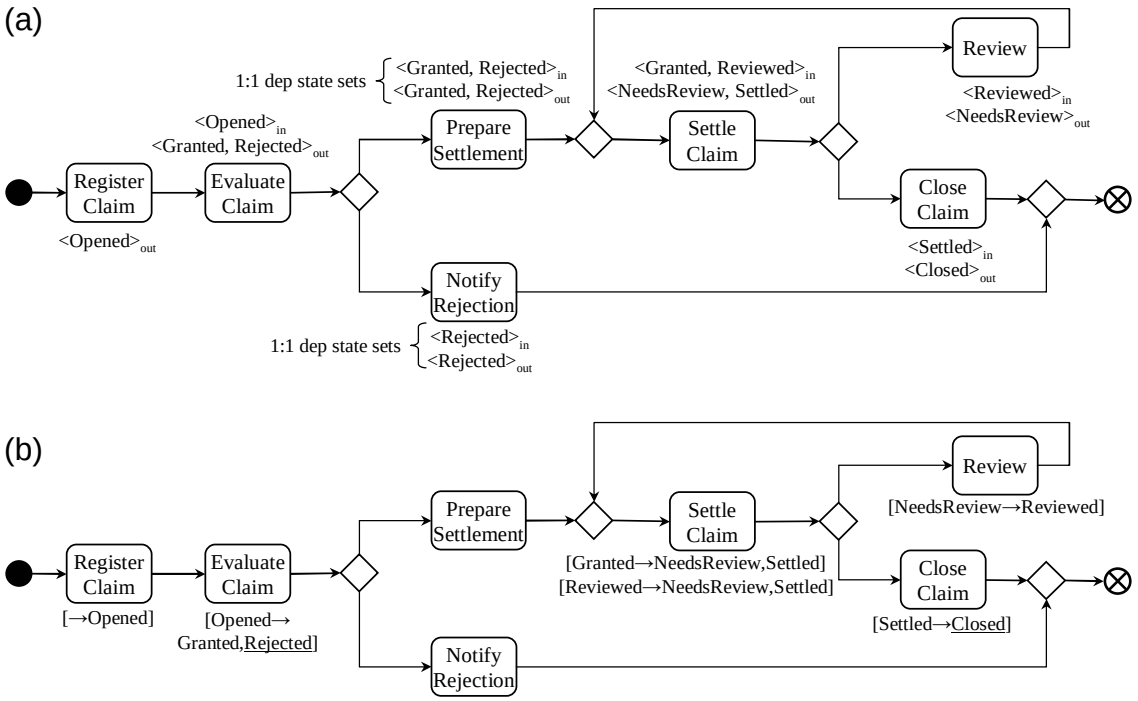


Figure 4.10: Induced transitions, first and last states

In Figure 4.10(b), first states $s_1, ..., s_n$ are indicated with an annotation $[\rightarrow s_1, ..., s_n]$, while last states are underlined. In this example, $Opened$ is the only first state, while $Closed$ and $Rejected$ are the last states of the $Claim$ object type.

We now define syntactic consistency conditions for the consistency concept for process and object life cycle models. These conditions can be evaluated directly over the elements of a given object life cycle model and the induced transitions, first and last states, derived from a given process model.

Definition 55 (Syntactic consistency conditions)**.** *Let a workflow graph $G = (N, E)$ with either repository or routed data flow using a set of object types T and a state specification be given. Furthermore, let an object life cycle model $OLC_t = (S, s_\alpha, s_\omega, \Sigma, \delta)$ be given. We define the following consistency conditions for G and OLC_t:*

C55.1: (transition conformance) for each induced transition (a, s_1, s_2) of t in G, there exists a transition from s_1 to s_2 in OLC_t;

C55.2: (first state conformance) for each first state s of t in G, there exists a transition from s_α to s in OLC_t;

C55.3: (last state conformance) for each last state s of t in G, there exists a transition from s to the final state s_ω in OLC_t;

C55.4: (transition coverage) for each transition from a state s_1 to a state s_2 in OLC_t where $s_1 \neq s_2$, $s_1 \neq s_\alpha$ and $s_2 \neq s_\omega$, there exists an induced transition (a, s_1, s_2) of t in G for some action $a \in N$;

C55.5: (initial transition coverage) for each transition from s_α to a state s in OLC_t, s is a first state of t in G;

C55.6: (final transition coverage) for each transition from a state s to s_ω in OLC_t, s is a last state of t in G.

The process and object life cycle models shown respectively in Figure 4.10(b) and Figure 4.9(b) do not satisfy several of these syntactic consistency conditions. For instance, transition conformance (C55.1) does not hold, since $(Settle\ Claim, Granted, NeedsReview)$ is an induced transition of $Claim$ in the shown process model, but $(Granted, NeedsReview)$ is not a transition in the given object life cycle model. Last state conformance (C55.3) does not hold either, since there is no transition from state $Rejected$ to the final state in the object life cycle model, but $Rejected$ is a last state of $Claim$ in the process model. Additionally, transition coverage (C55.4) is not satisfied, since there is no induced transition from state $Rejected$ to state $Closed$ of $Claim$ in the process model.

This example illustrates how the evaluation of syntactic consistency conditions allows us to determine process and object life cycle consistency according to its definition on the semantic level. In the following, we show that the syntactic consistency conditions are necessary and sufficient with regards to the definition of a process and object life cycle consistency for our selected class of process models.

4.3.4 Necessity and Sufficiency of Syntactic Consistency Conditions

In the following, we stipulate and prove the relation between the syntactic consistency conditions (Definition 55) and the process and object life cycle consistency (Definition 52).

Theorem 3. *Let an object life cycle model $OLC_t = (S, s_\alpha, s_\omega, \Sigma, \delta)$ and a workflow graph $G = (N, E)$ with repository data flow using a set of object types T be given such that N contains no forks (P3.1), and G satisfies properties P1, P2 and has a correct state specification. Syntactic consistency conditions in Definition 55 hold for G and OLC_t if and only if these models are consistent according to Definition 52.*

We introduce the following lemma to simplify the proof of Theorem 3:

Lemma 5. *Let a workflow graph $G = (N, E)$ with properties stated in Theorem 3 be given. Let a set of state sequences $Q_t = \{\langle s_{11}, ..., s_{1x}\rangle, ..., \langle s_{n1}, ..., s_{nz}\rangle\}$ for object type $t \in T$ generated by all possible executions of G be given. There exists a state sequence $q \in Q_t$ such that $\langle s_1, s_2\rangle$ is a subsequence of q (L5.1) if and only if $(a, s_1, s_2) \in induced(t)$ for some action $a \in N$ (L5.2).*

Proof. In the following, we prove this lemma in two parts.

L5.1 $\Rightarrow$ L5.2: We use proof by contradiction to show this. Let us assume that there exists a state sequence $q \in Q_t$ such that $\langle s_1, s_2\rangle$ is a subsequence of q. As the original hypothesis, we suppose that there is no action $a \in N$ such that $(a, s_1, s_2) \in induced(t)$. Since $\langle s_1, s_2\rangle$ is a subsequence of q, there must be an execution sequence $(w, O, s, h) \xrightarrow{a} (w', O', s', h')$ such that there is an object $o \in O \cap O'$ of type t, $s(o) = s_1$ and $s'(o) = s_2$. By following the same reasoning used to contradict H1 in the proof of Theorem 1, we can show that $s(o) \in \mathit{effin}(a, t)$ and hence $s_1 \in \mathit{effin}(a, t)$. Since a was executed according to Definition 49, we know that $s_2 \in dep(a, t, s_1)$. Since the $\frown$ operation does not append redundant states to state histories, we also know that $s_1 \neq s_2$. By the definition of induced transitions, this means that $(a, s_1, s_2) \in induced(t)$, which is a contradiction.

L5.2 $\Rightarrow$ L5.1: We use proof by contradiction to show this. Let us assume that there is an action $a \in N$ such that $(a, s_1, s_2) \in induced(t)$. As the original hypothesis, we suppose that there does not exist a state sequence $q \in Q_t$ such that $\langle s_1, s_2\rangle$ is a subsequence of q. Since

G satisfies properties P1, P2 and has a correct state specification, we know that a must be activated in some execution state (w, O, s, h). By the definition of induced transitions, we know that $s_1 \in \mathit{effin}(a,t)$, $s_2 \in \mathit{effout}(a,n,t)$ for some node $n \in N$ and $s_2 \in dep(a,t,s_1)$. By following the same reasoning used to contradict H2 in the proof of Theorem 1, we can show that there is an object $o \in O$ of type t such that $s(o) = s_1$. Since $s_2 \in dep(a,t,s_1)$, it is possible that action a is executed to change the state of object o to s_1 according to Definition 49. Therefore, $\langle s_1, s_2 \rangle$ must be a subsequence of some state sequence $q \in Q_t$, which is a contradiction. □

Proof of Theorem 3. Using Lemma 5, it is straightforward to show that C55.1 ⇔ C40.1 and C55.4 ⇔ C41.1 for states $s_1 \neq s_\alpha$ and $s_2 \neq s_\omega$, because for such states there is a one-to-one relation between subsequences $\langle s_1, s_2 \rangle$ of the state sequences generated by G and the induced transitions (a, s_1, s_2) of G. Another lemma can be defined similarly to Lemma 5, to prove the relation between the remaining conditions in Definition 55 and those in Definitions 40,41, which would cover the cases where $s_1 = s_\alpha$ and $s_2 = s_\omega$. □

Theorem 4. *Let an object life cycle model $OLC_t = (S, s_\alpha, s_\omega, \Sigma, \delta)$ and a workflow graph $G = (N, E)$ with routed data flow using a set of object types T be given (P3.1) such that G satisfies properties P1, P2 and has a correct state specification. Syntactic consistency conditions in Definition 55 hold for G and OLC_t if and only if these models are consistent according to Definition 52.*

Theorem 4 can be proven using the same line of argumentation as for Theorem 3.

In the presented definitions of the syntactic correctness and consistency conditions, effective states of objects in a process model play a significant role. In Section 4.2.3, a definition for effective input and output states was given without computation details. In the following section, we show how effective input and output states are computed according to this definition.

4.4 Computing Effective States in Process Models

Effective input and output states represent possible states of objects at different times during the execution of a process model. Computing these is thus a special type of a data-flow analysis problem, known from compiler theory (see e.g. [Muchnick, 1997]). In this section, we describe how effective states are computed based on an existing technique for data-flow analysis. We begin by presenting the basic concepts of data-flow analysis.

4.4.1 Data-Flow Analysis

Several data-flow analysis techniques have been developed in compiler theory to compute the possible run-time variable values for a given program. A *control-flow graph*, similar to a workflow graph, is generally used as a basis for performing data-flow analysis. Each node in a control-flow graph is associated with data-flow equations. At the beginning of the analysis, only a subset of the input and output values for the data-flow equations is known and the rest of the values are initialized to some default.

The most widespread approach to data-flow analysis is based on an iterative algorithm that repeatedly solves the data-flow equations and is thus referred to as *iterative data-flow analysis* [Kildall, 1973, Kam and Ullman, 1976]. For each node, output data values are computed based on the input data values and then propagated to the input data values of

the successor nodes. In forward flow analysis, where data values are propagated downstream in the control-flow graph, a *reverse postorder traversal* of the nodes ensures that in the absence of cycles each node is only visited once. In the presence of cycles, the nodes are traversed repeatedly until a *fixpoint* is reached, i.e. an iteration when no input or output data values change.

The data-flow equations for a given node n are defined in the following form:

- $out_n = trans_n(in_n)$
- $in_n = join_{p \in pred_n}(out_p)$

The $trans_n$ function is referred to as the *transfer function* of node n, which produces an output data value out_n based on an input data value in_n. The *join operation* $join_{p \in pred_n}$ yields an input data value in_n for node n based on the output data values of the predecessor nodes of n.

The iterative data-flow analysis converges to a fixpoint under the following conditions: the data value domain must be a partial order with a finite height, and the transfer function and the join operation must be monotonic with respect to this partial order. This ensures that on each iteration a data value either stays the same or grows larger (monotonicity) without growing indefinitely large (finite height).

Given a control-flow graph $G = (N, E)$, Kam and Ullman [Kam and Ullman, 1976] have shown that visiting the nodes N in a reverse postorder using an iterative data-flow analysis requires $d(G) + 3$ passes over the graph G in the worst case. $d(G)$ is the depth or loop-connectedness of G, which is the maximum number of back edges on an acyclic path of G with respect to a Depth-First Spanning Tree (DFST) of G. Each pass over the graph requires an iteration over $|N|$ nodes. Assuming that computing the join operation also requires an iteration over $|N|$ nodes in the worst case, the iterative data-flow analysis performs a total of $(d(G) + 3) \times |N| \times |N|$ iterations. Therefore, the complexity of the iterative data-flow analysis is $O(d(G) \times |N|^2)$ or $O(|E|^3)$. In practice, however, d is often 3 or less [Knuth, 1971, Kam and Ullman, 1976]. The efficiency of the join operation can also be greatly improved by using *bit vectors* to represent the set of data definitions that reach different nodes in the control-flow graph, which allows the join to be implemented as a set of bitwise logical operations [Kildall, 1973]. In the following, we show how basic iterative data-flow analysis (without bit vectors) can be used to compute effective states.

4.4.2 Using Iterative Data-Flow Analysis to Compute Effective States

In the context of computing effective states, the data value domain is $\mathcal{P}(S_T)$ that comprises sets of object type states. This domain is a powerset and thus a partial order of a finite height with respect to the $\subseteq$ relation. Given an object type t, the data-flow equations for an action or a decision n are as follows: $out_{n,t} = \mathit{effout}(n, n_q, t)$ where $n \lhd_t n_q$ and $in_{n,t} = \mathit{effin}(n, t)$. These are both monotonic, as by definition states are only added to and never removed from the effective input and output state sets, as more information becomes available.

By applying the basis of the iterative data-flow analysis technique, the computation of the effective states is achieved using the algorithm described in the pseudocode in Listing 4.1. The algorithm computes effective input and output states of each action and decision for each object type in the given process model. First, the effective input and output state sets are initialized to empty sets for each action and decision (line 4). The object provider graph is then computed based on the object provider relation between the actions and decisions for the current object type (line 5). A reverse postorder is then computed

on the object provider graph (line 6), which is not detailed here as this can be done by reversing a postorder achieved by a depth-first search of the graph. The actions and decisions using the current object type are then repeatedly traversed in this reverse postorder, evaluating the effective input states of each action or decision, until a fixpoint is reached (lines 9-14). The effective input and output states are evaluated according to Definition 46, as described in the remainder of the pseudocode (lines 16-45).

Listing 4.1: computeEffectiveStates

```
1  computeEffectiveStates(ProcessModel p)
2    for each (ObjectType t in p.getObjectTypes()) do
3      // initialize and prepare object provider graph
4      initializeEffectiveStates (p,t);
5      Graph objectProviderGraph = computeObjectProviderGraph(p,t);
6      Order orderToTraverse = computeReversePostorder(objectProviderGraph);
7      boolean fixpointReached = false;
8      // iterate over object provider graph until fixpoint
9      while (!fixpointReached) do
10       boolean change = false;
11       while (orderToTraverse.hasMoreElements()) do
12         ActionDecision ad = orderToTraverse.next();
13         change = evaluateEffectiveInStates (t,ad);
14       fixpointReached = !change;
15
16 evaluateEffectiveInStates (ActionDecision ad, ObjectType t)
17   StateSet newEffectiveInStates = emptySet;
18   for each (ActionDecision op in getObjectProviders(ad)) do
19     newEffectiveInStates .addAll( getEffectiveOutStates(op,ad,t);
20   // determine whether new effective input states were added
21   if (newEffectiveInStates == getEffectiveInStates(t,ad)) then
22     return false;
23   else
24     setEffectiveInStates (t,ad, newEffectiveInStates );
25     return true;
26
27 evaluateEffectiveOutStates (ActionDecision op, ActionDecision ad, ObjectType t)
28   StateSet newEffectiveOutStates = emptySet;
29   // case 1: action with no such data input
30   if (isAction(op) & !op.getDataInputs(). contains(t))
31     newEffectiveOutStates = op.producedStates();
32   // case 2: action that has such data input
33   if (isAction(op))
34     for each (State s in evaluateEffectiveInStates (op, t)) do
35       newEffectiveOutStates.addAll(getDependencyStateSet(t, op, s));
36   // case 3: decision
37   else if (isDecision(op))
38     for each (Edge e in op.outEdgesLeadingTo(ad)) do
39       newEffectiveOutStates.add(e.getConditionStates());
40   // determine whether new effective output states were added
41   if (newEffectiveOutStates == getEffectiveOutStates(op,ad,t)) then
42     return false;
43   else
44     setEffectiveOutStates (o,ad, newEffectiveInStates );
45     return true;
```

Returning to one of our previous examples shown in Figure 4.11 (cf. Figure 4.4), we can see how the presented technique can be used to compute effective states for a given process model. Figure 4.11(b) shows the object provider graph computed for object type C ($Claim$). In this object provider graph, all effective input and output states are initialized to empty sets. The final result of the computation is shown in Figure 4.11(c). In this example, the computation of effective states requires only one iteration over the object provider graph, since there are no cycles in this graph.

Based on the computation of effective states in a given process model, the syntactic

correctness and consistency conditions defined in Sections 4.2.3 and 4.3.3 can be evaluated in a straightforward manner.

The overall approach to the evaluation of the syntactic correctness and consistency conditions represents a *sound* and *complete* method for determining state specification correctness and process and object life cycle consistency, with respect to the selected class of process models. Soundness in this context (not to be confused with soundness of control flow) means that evaluation of our conditions does not produce false-negatives. An example of a false-negative is a situation where the syntactic correctness conditions are not satisfied, but the state specification is indeed correct. On the other hand, completeness means that no false-positives are produced by the evaluation of the conditions. In the case of state specification correctness, this means that whenever the state specification is incorrect, the syntactic correctness conditions are not satisfied.

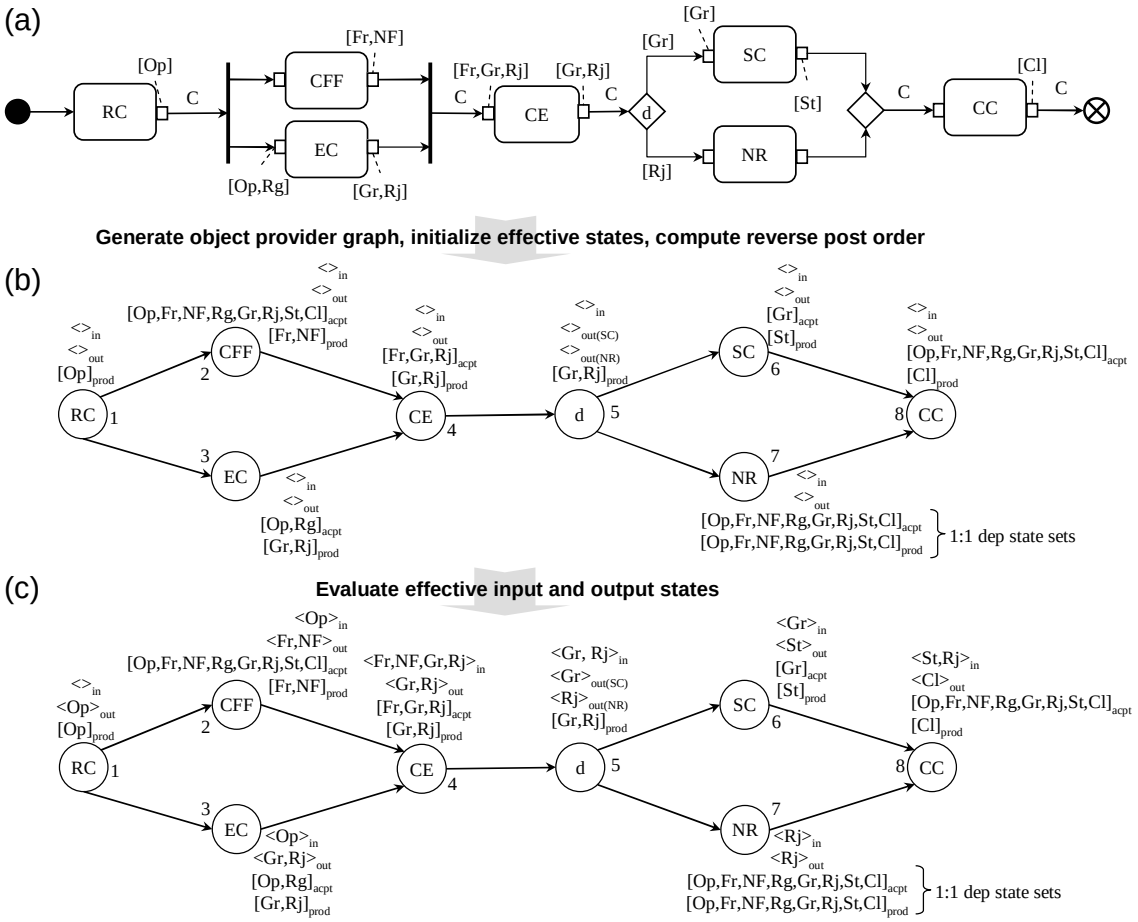


Figure 4.11: Example evaluation of effective input and output states

The syntactic correctness and consistency conditions can also be evaluated for process models outside the selected class. In the case of process models with repository data flow that contain forks, but are at the same time sound with regards to control flow and have no missing data, the evaluation produces an approximation of the state specification correctness and process and object life cycle consistency. For such process models, it is possible that in some cases the evaluation produces false-negatives and false-positives. In the following, we discuss alternative approaches that could be used for consistency checking in the context of process and object life cycle models.

4.4.3 Alternative Approaches to Consistency Checking

The consistency definitions that we presented in this chapter, i.e. state specification correctness and process and object life cycle model consistency, are not coupled with any particular approach to consistency checking. We tackled consistency checking by defining syntactic conditions that can be decided using iterative data-flow analysis. On the one hand, this approach is straightforward to implement and integrate into existing tools, it is efficient for the general case, and it is capable of providing the modeler with diagnostic information in case of condition violation. On the other hand, the approach is not sound or complete for process models with repository data flow that contain forks. Other analysis techniques could potentially be used for consistency checking instead of or in combination with the demonstrated approach, as described next.

Model checking [Clarke et al., 1999] is an automatic analysis technique for verifying models of finite state concurrent systems. As opposed to checking a model statically as in data-flow analysis, model checking involves an exhaustive search of the execution state space of a given model to determine whether a given property is satisfied. The property and model under consideration are usually expressed in temporal logic and a tool-specific language (for tools such as NuSMV [Cimatti et al., 2002] or SPIN [Holzmann, 2003]), respectively. Since models that express highly concurrent behavior lead to very large execution state spaces, model checking is known to suffer from the so-called *state space explosion* problem. A number of techniques, including symbolic algorithms using binary decision diagrams [McMillan, 1993] and partial order reduction [Valmari, 1992, Peled, 1994], have been developed to improve the efficiency of model checking.

Several applications of model checking for analyzing business process models have been described in the literature. For example, Janssen et al [Janssen and Mateescu, 1998, Janssen et al., 1999] describe verification of process models in a language called Amber against user-defined temporal properties using the SPIN model checker. Amber process models are based on the same control-flow constructs as workflow graphs and have no data flow. The authors demonstrate the feasibility of the approach on one example process model of a relatively small size. Further examples of process model verification using SPIN are reported in [Matoušek, 2003]. Eshuis and Wieringa [Eshuis and Wieringa, 2004] apply model checking to UML Activity Diagrams using an extended version of NuSMV. In process models under considerations, data flow is represented implicitly by variables that are used to define edge conditions. The authors propose several reduction rules to fight the state space explosion problem and demonstrate on an industrial case study that the reduction significantly decreases the analysis time. Two more case studies performed using this approach are described by Eshuis in his PhD dissertation [Eshuis, 2002]. The NuSMV model checker has also been employed for the verification of business process models against quality constraints [Förster et al., 2007].

Existing works that use model checking for the analysis of process models can potentially be extended to decide consistency described in this chapter, i.e. to determine whether a state specification in a given process model is correct or whether a given process model and a given object life cycle model are consistent. This would require that an existing mapping of process models to an input language of a model checker is extended to take into account repository and routed data flow. Furthermore, the correctness of a state specification and process and object life cycle model consistency (cf. Definitions 44 and 52) would need to be expressed in temporal logic. Adding data flow would certainly exacerbate the performance of model checking process models and can potentially result in a significant delay for process models that manipulate a large number of object types with many states.

Analysis approaches based on Petri nets [Peterson, 1981, Murata, 1989] can also be used to check properties of models that represent concurrent behavior. Apart from an exhaustive search of the reachable execution states, techniques based on matrix equations and reduction are employed in Petri net analysis to decide certain properties faster. In [van der Aalst et al., 2002], van der Aalst et al describe the analysis of process models by mapping them to a subclass of Petri nets, called workflow nets. The authors show that control-flow soundness can be decided in polynomial time for workflow nets that satisfy the free-choice property[1]. This approach is implemented in the Woflan tool [Verbeek et al., 2001], which has been validated to produce results in a reasonable amount of time for significantly sized process models. The analysis implemented in Woflan abstracts from data flow.

The works by Stoerrle [Stoerrle, 2005] and Mendling et al [Mendling et al., 2008] are the only ones we are aware of that explicitly handle data flow in process model analysis using Petri nets. Stoerrle formalizes the semantics of UML Activity Diagrams with data flow using colored Petri nets (CPN) [Jensen, 1995]. In [Stoerrle, 2005], verification against standard properties, such as proper termination and the absence of deadlocks, and user-defined properties is discussed. Mendling et al provide an algorithm for generating a reachability graph from a process model in the EPCs notation extended with object flows, which enables the analysis of such process models for soundness. The authors state the relationship between control-flow soundness and soundness of an extended EPC.

The recent work by Vanhatalo et al [Vanhatalo et al., 2007, Vanhatalo et al., 2008] enables the analysis of process models using a combination of fast static analysis techniques that may be incomplete with more expensive full verification approaches. Process models are decomposed into so-called *Single-Entry-Single-Exit (SESE) fragments*, which facilitates the analysis of each fragment in isolation. The described analysis [Vanhatalo et al., 2007] focuses on control-flow soundness and does not consider data flow. Using industrial case studies, the authors have shown that soundness can be decided using static analysis for a large percentage of fragments that occur in realistic process models. Fragments with undecided soundness can be analyzed using a complete analysis technique such as model checking, as described by Favre [Favre, 2008].

Following the principles of the works by Vanhatalo et al and Favre, we could apply a hybrid analysis approach based on fragment decomposition for consistency checking in the context of process and object life cycle models. Fragments comprising routed data flow or repository data flow with no forks could be analyzed based on iterative data-flow analysis, as described in this chapter, while model checking or another complete analysis technique could be applied to other fragments. The realization of such an approach is outside the scope of this dissertation.

4.5 Summary and Discussion

In this chapter, we addressed the intra-model and inter-model consistency aspects of process and object life cycle models. The first part of the chapter was concerned with the correctness of a state specification in a process model. We first defined the notion of correctness in terms of the process model semantics. Subsequently, we introduced structural correctness conditions that guarantee the correctness of a state specification for process models with routed data flow and for process models with repository data flow that contain no forks.

[1] In a free-choice Petri net, every output transition t of a place s is either the only output transition of s or s is the only input place of t. See [Desel and Esparza, 1995] for further reading.

In the second part, we established a consistency concept for process and object life cycle models. We first brought these model types together by defining their relation to a common semantic domain of object state sequences, which we then used to define consistency on the semantic level. We then defined a set of syntactic consistency conditions that guarantee semantic process and object life cycle consistency for process models with routed data flow and for process models with repository data flow that contain no forks.

Evaluation of syntactic correctness and consistency conditions is largely reliant on the computation of the so-called effective states in a given process model. We described how effective states can be computed using a static analysis technique based on iterative data-flow analysis. Furthermore, we discussed a range of alternative approaches to computing effective states and hence checking consistency.

Chapter 5

Inconsistency Resolution

This chapter is concerned with the resolution of model inconsistencies. We begin by giving an overview of the main concepts and challenges of inconsistency resolution in Sections 5.1 and 5.2, respectively. After that, we present our proposed solution to address these challenges in Sections 5.3-5.6. While our solution can be used to alleviate inconsistency resolution for different types of models, we illustrate the solution using our focus domain of process and object life cycle models. At the end of the chapter, we discuss our solution in the broader context of inconsistency management in Section 5.7.

5.1 Main Concepts of Inconsistency Resolution

"Inconsistency" and "resolution" are naturally the key concepts identified in the area of inconsistency resolution. However, depending on the context in which these terms are used, their interpretations may differ. For instance, inconsistency may refer to the overall situation where model consistency does not hold or it may refer to one particular consistency condition not being satisfied. On the other hand, resolution may refer to a model transformation that ensures that a particular consistency condition holds or it may refer to an actual application of such a model transformation. As a clarification, we distinguish between the *type level* and the *instance level* in inconsistency resolution, as illustrated in Figure 5.1. For each level, the associated concepts and roles are displayed. This diagram distills the concepts described in the existing literature, such as [Nentwich et al., 2003, Van Der Straeten, 2005, Mens and Van Der Straeten, 2006, Egyed, 2007].

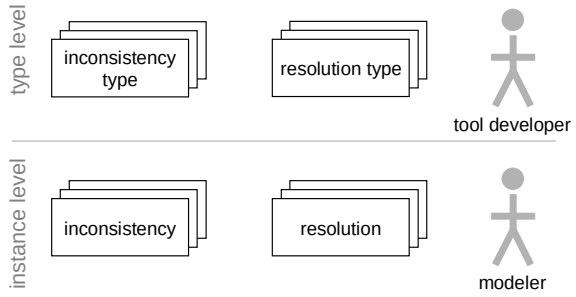


Figure 5.1: Type level and instance level in inconsistency resolution

The type level is concerned with concepts defined in terms of modeling languages, while the instance level is concerned with concepts defined in terms of concrete models.

5.1.1 Inconsistency Type and Inconsistency

An *inconsistency type* is an abstract description of a violation of a particular consistency condition. On the other hand, an *inconsistency* is a concrete violation of some consistency condition in a given set of models. Therefore, an inconsistency can be seen as an instance of an inconsistency type. As opposed to an inconsistency type, an inconsistency has a *context* comprising concrete model elements that contribute to the violation of the consistency condition.

In Chapter 4, we defined a set of syntactic consistency conditions for checking process and object life cycle model consistency (see Definition 55). Based on these conditions, we define six inconsistency types. We assume that a workflow graph $G = (N, E)$ with repository or routed data flow using a set of objects T and a state specification, and an object life cycle model $OLC_t = (S, s_\alpha, s_\omega, \Sigma, \delta)$ for object type $t \in T$ are given. Inconsistency types are defined as follows:

- *non-conformant transition:* there is an induced transition (a, s_1, s_2) of t in G, but no transition from state s_1 to state s_2 in OLC_t;
- *non-conformant initial transition:* there is a first state s_{first} of t in G, but no transition from the initial state to state s_{first} in OLC_t;
- *non-conformant final transition:* there is a last state s_{last} of t in G, but no transition from s_{last} to the final state in OLC_t;
- *non-covered transition:* there is a transition from state s_1 to state s_2 in OLC_t where s_1 is not the initial state, but no induced transition (a, s_1, s_2) of t in G for any action a;
- *non-covered initial transition:* there is a state s_i in OLC that has an incoming transition from the initial state, but s_i is not a first state of t in G;
- *non-covered final transition:* there is a state s_f in OLC that has an outgoing transition to the final state, but s_f is not a last state of t in G;

Contexts of inconsistencies are derived from the definitions of their inconsistency types given above. A non-conformant transition inconsistency is denoted $ncnf\_tran(a, s_1, s_2)$, with action a and states s_1, s_2 in its context. A non-conformant initial transition inconsistency is denoted $ncnf\_init(a, s_{first})$, where a is the action that produces the first state s_{first}. A non-conformant final transition inconsistency is written as $ncnf\_fin(n, s_{last})$, where n is the node that produces the last state s_{last}. Furthermore, a non-covered transition is denoted $ncov\_tran(s_1, s_2)$, a non-covered initial transition inconsistency as $ncov\_init(s_i)$, and a non-covered final transition inconsistency as $ncov\_fin(s_f)$.

5.1.2 Resolution Type and Resolution

A *resolution type* is an abstract description of how inconsistencies of a particular inconsistency type can be resolved. On the other hand, a *resolution* is an operation on the elements of a concrete set of models, which resolves a particular inconsistency. A resolution can be seen as an instance of a resolution type. Since a resolution is associated with concrete model elements, it also has a context.

A variety of resolution types can be defined to address the six inconsistency types defined for process and object life cycle models (cf. Section 5.1.1). Three sample resolution types for removing non-conformant transition inconsistencies are listed below and

illustrated in Figure 5.2[1]. These resolution types assume a non-conformant transition inconsistency $ncnf\_tran(a, s_1, s_2)$ with (a, s_1, s_2) being an induced transition in the given process model.

(a) rt_1 removes state s_1 from $acpt(a, t)$, removing the entire data input of type t if s_1 is the last state in $acpt(a, t)$;

(b) rt_2 removes s_2 from $prod(a, t)$, removing the entire data output of type t if s_2 is the last state in $prod(a, t)$;

(c) rt_3 removes the entire action a from the process model.

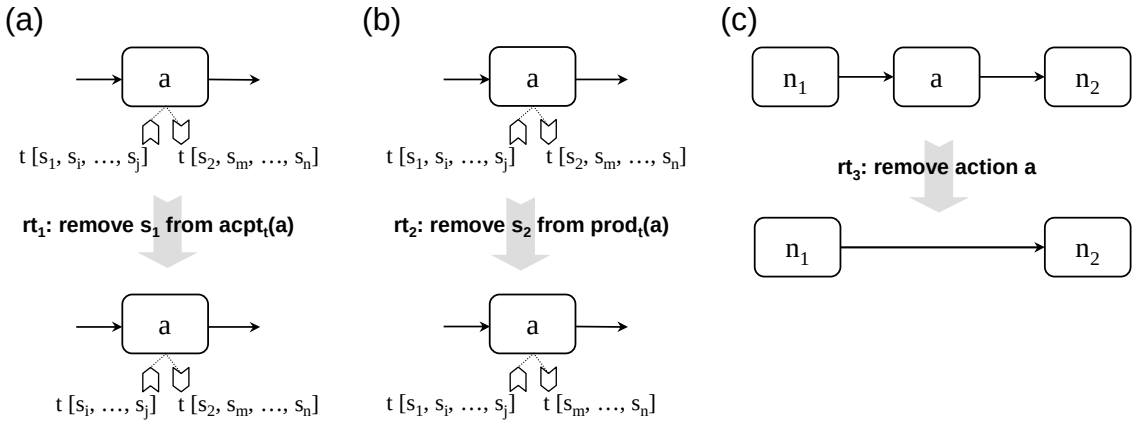


Figure 5.2: Resolution types for non-conformant transition inconsistencies

Suppose that an inconsistency $ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$ was detected between a claims handling process model and an object life cycle model for type $Claim$. An example of applying resolutions of types rt_1, rt_2, rt_3 to this inconsistency is shown in Figure 5.3. These can be considered as instances of the resolution types rt_1, rt_2, rt_3.

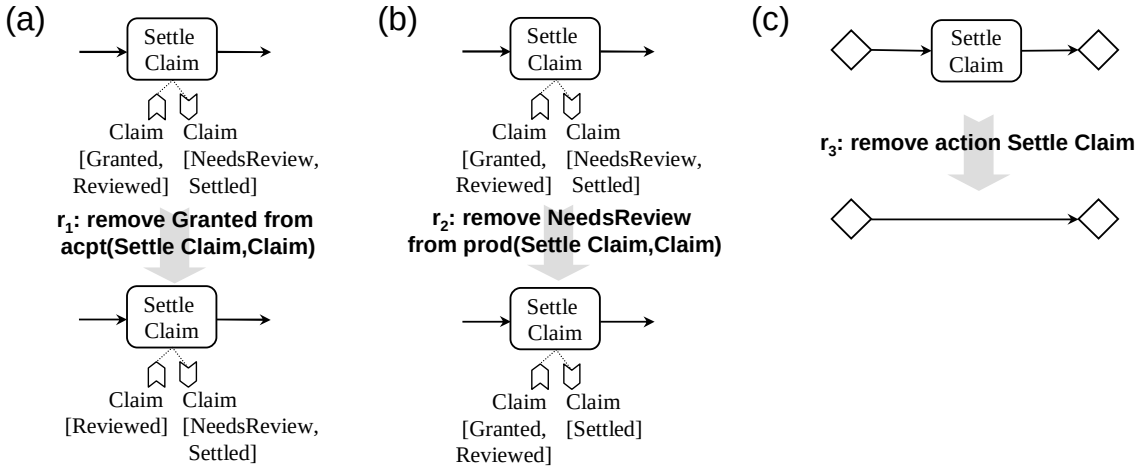


Figure 5.3: Example resolutions

[1]We use informal diagrams, which could be refined into more precise specifications in a model transformation language (e.g. using VIATRA [Csertán et al., 2002] or GReAT [Agrawal, 2004]).

5.1.3 Formalizing Main Concepts

We next formalize the introduced type-level and instance-level concepts. For the purpose of the formalization, we consider inconsistency resolution in view of *one* model, which may comprise several *submodels*. This approach allows us to address resolution of intra-model and inter-model inconsistencies at the same time. Each model and submodel comprise a set of atomic model elements. For example, if we are concerned with the resolution of process and object life cycle inconsistencies, we consider these inconsistencies in the context of one model comprising two submodels: a process model and an object life cycle model.

Definition 56 (Model and submodel)**.** *A* model $M = \{me_1, ..., me_n\}$ *is defined as a set of model elements* $me_1, ..., me_n$*. Each model* M *comprises one or more* submodels $M_1, ..., M_p$*, where each submodel is itself a model and* $M = \bigcup_{i=1}^{p} M$*.*

A model is associated with inconsistency types and each inconsistency type is associated with a set of resolution types, where the internals of inconsistency and resolution types are not further specified.

Definition 57 (Inconsistency type and resolution type)**.** *Given a model* M*, we denote the set of* inconsistency types *defined for a model* M *as* $\mathcal{I}_M$*, where each inconsistency type* $it \in \mathcal{I}_M$ *is associated with a set of* resolution types $\mathcal{R}_{it}$*.*

An inconsistency is defined in terms of its inconsistency type and its context. In the following, we distinguish between the *model context* and the *model element context* of an inconsistency.

Definition 58 (Inconsistency, model context, model element context)**.** *Given a model* $M = \{me_1, ..., me_n\}$*, we denote the set of* inconsistencies *in* M *as* I_M*. Each inconsistency* $i \in I_M$ *has a type* $it \in \mathcal{I}_M$*, contains the model* M *in its* model context *and has a non-empty* model element context $\{me_j, ...me_k\} \subseteq M$ *of model elements that contribute to this inconsistency. We write* $i = it(me_j, ..., me_k)$*.*

As expected, a resolution is defined to resolve a particular inconsistency. A resolution is defined in terms of its resolution type and the context of the inconsistency it resolves.

Definition 59 (Resolution)**.** *A given inconsistency* $i = it(me_j, ..., me_k) \in I_M$ *in a model* M*, is associated with a set of* resolutions R_i*. Each resolution* $r_i \in R_i$ *has a type* $rt \in \mathcal{R}_{\rangle\sqcup}$*. We write* $r_i = rt(me_j, ..., me_k)$*.*

Since inconsistency resolution is predominantly tool-supported, we identify the *tool developer* as the role responsible for defining inconsistency and resolution types, as illustrated in Figure 5.1. In turn, the *modeler* using a particular tool for inconsistency resolution is concerned with handling inconsistencies, and choosing among and applying different resolutions. In this chapter, we focus on the main challenges facing the modeler during inconsistency resolution, which are described in the following section.

5.2 Challenges of Inconsistency Resolution

After performing a consistency check on a set of given models, the modeler is confronted with a set of detected inconsistencies that need to be resolved. The modeler typically proceeds by choosing one inconsistency, understanding its context, analyzing the available resolutions for that inconsistency and then applying one of these resolutions. Unless

some inconsistencies can be tolerated, the modeler repeatedly applies these steps until all inconsistencies have been resolved. We identify four main challenges facing the modeler during inconsistency resolution, as described next.

5.2.1 Inconsistency Prioritization and Context-Switching

The first challenge confronting the modeler is to determine the order in which inconsistencies should be processed, in other words *inconsistency prioritization*. Inconsistency prioritization can be seen as defining a partial order on a set of given inconsistencies to optimize the overall resolution of the given inconsistencies in some way. The optimization can be targeted to minimize the modeler effort required during the resolution or to ensure the satisfaction of some domain-specific requirements.

Existing work in the area of inconsistency management mentions the issue of inconsistency prioritization [Spanoudakis and Zisman, 2001], but does not address it in much detail. However, some of the performed experiments in this area report dealing with up to 10,465 inconsistencies [Egyed, 2007]. While in this work, Egyed evaluates his proposed tool support for inconsistency resolution, this tool support does not incorporate inconsistency prioritization. Dealing with such a large number of inconsistencies without prioritization would, however, be extremely challenging. Realistic collections of process and object life cycle models, such as those in the IBM Insurance Application Architecture (IAA)[2], comprise a large number of complex models and can therefore also give rise to a large number of inconsistencies that would need to be prioritized.

Regardless of the order in which inconsistencies are processed, moving between inconsistencies always requires the modeler to switch between inconsistency contexts. However, handling the inconsistencies in an arbitrary order can lead to excessive *context-switching* for the modeler. This leads to an increased amount of time required for the inconsistency resolution for the following reasons: Context-switching in graphical modeling tools requires showing the relevant model editor and navigating to different parts of models for inconsistency visualization, which takes a noticeable amount of time even in the latest available tools. Furthermore, at each context-switch the modeler is required to spend time studying the new inconsistency context. Therefore, the modeler also needs to consider the context-switching aspect when prioritizing inconsistencies. To the best of our knowledge, no existing work in the area of inconsistency resolution addresses the problem of context-switching.

5.2.2 Resolution Impact Analysis

Another challenge facing the modeler is to analyze the overall impact that each resolution available for a given inconsistency induces if it is applied. The *resolution impact* includes domain-specific aspects resulting from how the resolution changes the underlying models and the overall impact on the set of inconsistencies in the given models.

Since resolutions may have side-effects (cf. Section 2.4.2 of Chapter 2), i.e. applying a resolution may add or remove inconsistencies other than the target inconsistency. Inconsistencies that are removed from or added to the current set of inconsistencies as a side-effect are respectively referred to as *expired* and *induced inconsistencies*. This is illustrated in Figure 5.4(a), where $i_2, ..., i_n$ are expired inconsistencies and $i_{n+1}, ..., i_p$ are induced inconsistencies. As a concrete example, consider that i_1 is the inconsistency $ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$ and the resolution applied is r_1

[2] `http://www.ibm.com/industries/financialservices/doc/content/solution/278918103.html`

shown in Figure 5.3(a). As a side-effect, this resolution may lead to an induced inconsistency $ncov\_tran(Granted, Settled)$ or to an expired inconsistency $ncnf\_tran(Settle\ Claim, Granted, Settled)$. The actual induced and expired inconsistencies depend on the models at hand and on the current set of inconsistencies. Therefore, resolution impact analysis requires the modeler to manually examine the models and the inconsistencies in detail if no automated support for this is provided.

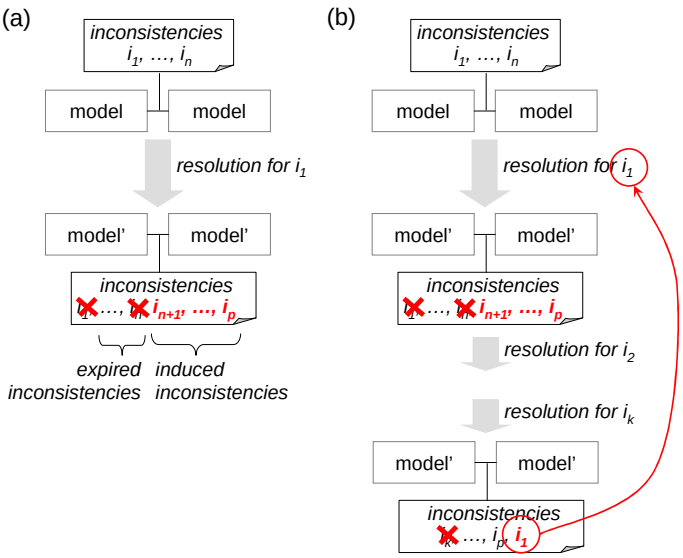


Figure 5.4: (a) Induced and expired inconsistencies (b) Resolution cycle

Providing automated support to assist the modeler in dealing with resolution side-effects has been the topic of several recent publications [Mens et al., 2006a, Mens et al., 2006b, Egyed, 2007]. These works analyze inconsistency and resolution types defined at the type level. Type-level analysis, for example using graph transformation tools [Mens et al., 2006a, Mens et al., 2006b], can identify dependencies between inconsistency and resolution types. On the instance level, such dependencies can be used to determine resolution side-effects that *can potentially occur* if a particular resolution is applied. Given a set of inconsistency types and resolution types with many dependencies, there will be many potential side-effects for each resolution at the instance level. While this information can be helpful to the modeler during the impact analysis of resolutions, it requires the modeler to examine all the potential side-effects each time to determine which ones really occur for the given resolution.

5.2.3 Comparison of Alternative Resolutions

Since several alternative resolutions are often available for one inconsistency, the modeler needs to draw a comparison, weighing the advantages and disadvantages of each resolution. For example, one of the three alternative resolutions shown in Figure 5.3(a), (b) and (c) can be applied to resolve the inconsistency $ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$. In many cases, it may not be immediately clear which of the available resolutions is the most beneficial one.

Supporting the modeler in the comparison of alternative resolutions is emphasized in [Spanoudakis and Zisman, 2001]. The authors argue that resolutions should be ordered based on cost, risk and benefit. However, no extensive methods for providing such support to the modeler currently exist, to the best of our knowledge.

5.2.4 Avoidance of Resolution Cycles

The last challenge that we identify is the avoidance of resolution cycles. When applying a sequence of resolutions to different inconsistencies, the modeler may end up in a resolution cycle if a previously resolved inconsistency is introduced by a later resolution as a side-effect (cf. Section 2.4.2 of Chapter 2). This is illustrated in Figure 5.4(b), where i_1 is removed by the first resolution and is added as an induced inconsistency by the last resolution. Suppose that i_1 is the inconsistency $ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$ and the first resolution is r_1 shown in Figure 5.3(a), which removes state $Granted$ from the accepted states of $Settle\ Claim$ for $Claim$. As a side-effect, r_1 may introduce the inconsistency $ncov\_tran(Granted, Settled)$, which could in turn be resolved by adding state $Granted$ to the accepted states of $Settle\ Claim$ for $Claim$. This would lead to the original inconsistency $ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$ being introduced and thus to a resolution cycle.

Similar to a resolution side-effect, a resolution cycle concerns inconsistencies in concrete models and therefore it is an instance-level concept. Some of the existing work tackling resolution side-effects also addresses resolution cycles [Mens et al., 2006b, Mens and Van Der Straeten, 2006]. A type-level analysis is used for detecting cycles, which represents a "conservative approximation of what can actually happen in a concrete setting" [Mens and Van Der Straeten, 2006]. Furthermore, there is work on incremental transformations using triple graph grammars [Schürr, 1994] studies the problem of keeping two models synchronized [Giese and Wagner, 2006, Becker et al., 2007]. This is achieved by analyzing changes in one model and applying incremental updates for reestablishing consistency. Cycles in the resolution process can be detected during automatic inconsistency resolution by storing and checking against an inconsistencies history. When an occurrence of a cycle is detected, the modeler is informed that manual intervention is required. Precise forecasting that a particular resolution leads to a resolution cycle is not supported in any of these approaches.

In the following four sections, we present our solution to address the four challenges facing the modeler during inconsistency resolution described above. First of all, we propose to prioritize inconsistencies in a way that minimizes context-switching required from the modeler and takes into account modeler-defined priorities that can capture domain-specific requirements (Section 5.3). Secondly, we assist the modeler in impact analysis of resolutions by forecasting resolution side-effects (Section 5.4). The forecast is enabled by the specification of so-called *side-effect expressions* for resolution types by the tool developer. Furthermore, the side-effect forecast is combined with *inconsistency costs* to quantify the relative advantages of resolutions, therefore assisting the modeler in the comparison of alternative resolutions (Section 5.5). Finally, to help the modeler in the avoidance of resolution cycles, we describe how resolutions can be placed into different categories based on whether they can lead to a resolution cycle or not (Section 5.6). Each part of the solution is demonstrated using our focus domain of process and object life cycle models, however the concepts of the solution can also be applied in other domains.

5.3 Inconsistency Prioritization to Minimize Context-Switching

In this section, we define several types of context-switches that the modeler can incur during inconsistency resolution and propose to prioritize a given set of inconsistencies such that the total number of all context-switches is minimized (cf. Section 5.2.1). An overview of the proposed inconsistency prioritization is shown in Figure 5.5. The original

set of detected inconsistencies is processed by multiple levels of *grouping* and *ordering*, at the end of which a final prioritized set of inconsistencies is obtained.

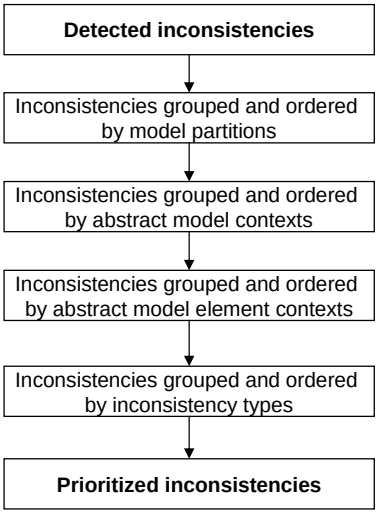


Figure 5.5: Inconsistency prioritization

Figure 5.5 shows that the concepts of a *model partition*, an *abstract model context* and an *abstract model element context* are employed in the first three steps of the grouping and ordering. In the following, we explain these concepts, define the different types of context-switches and then provide an algorithm for the inconsistency prioritization.

Given a model that comprises a set of submodels, a model partition identifies a subset of inter-related submodels, which are not related to any other submodels of the given model. Two models are considered related if they can occur in the model context of the same inconsistency. For example, we can assume that given a set of process and object life cycle models, inconsistencies are only reported for a pair of models (M_1, M_2), where M_1 is a process model that manipulates objects of the type for which the object life cycle model M_2 is defined. Consequently, a process model can be considered related to an object life cycle model for a particular object type only if objects of this type are manipulated in the given process model.

Definition 60 (Model partition)**.** *Let a model $M = \{M_1, ..., M_p\}$ comprising submodels $M_1, ..., M_p$ be given and a relation $\psi : M \times M$ on the submodels of M be given. Given an undirected graph (M, ψ) that has submodels as nodes and submodel relations as edges, a* model partition *is a maximal connected subgraph of this graph.*

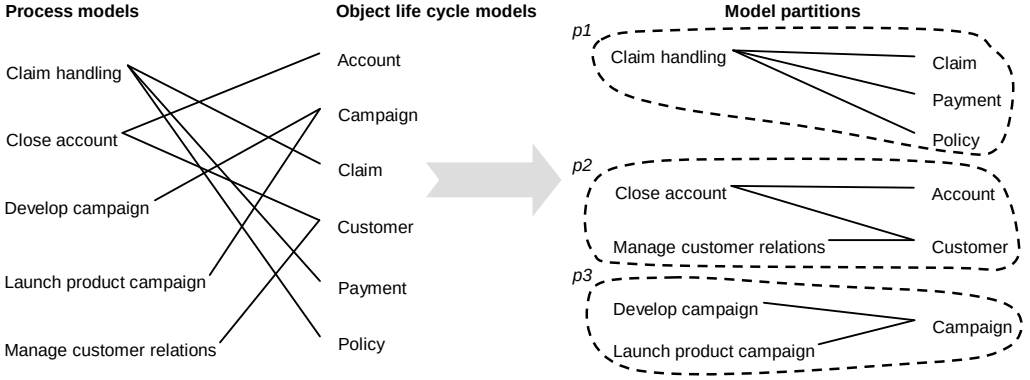
Let us consider a sample set of inconsistencies detected for a given set of process and object life cycle models, as shown in Figure 5.6(a). By considering the models in the model contexts of these inconsistencies and the relations between these models, three model partitions shown in Figure 5.6(b) are identified. The detected inconsistencies can be grouped according to these three model partitions, as shown in Figure 5.6(c). Such a grouping of inconsistencies is used in the overall inconsistency prioritization, as explained later in this section.

An abstraction of model contexts and model element contexts is performed in order to distinguish between different types of context-switches. Given an inconsistency, the most "coarse-grained" elements in its model element context are identified to comprise its abstract model element context. For example, abstract model element contexts can be defined as the elements of the model element context that have the most easily-identifiable

(a)

No.	Model context	Inconsistency category	Inconsistency type	Model element context
1	Claim handling, Claim	Conformance	Non-conformant transition	register, Received-Opened
2	Close account, Account	Conformance	Non-conformant transition	close, Opened-Closed
3	Claim handling, Payment	Conformance	Non-conformant transition	carry out settlement, Created-Paid in full
4	Close account, Customer	Conformance	Non-conformant transition	update customer status, Active-Dormant
5	Claim handling, Claim	Conformance	Non-conformant transition	close claim, Paid-Closed
6	Claim handling, Payment	Coverage	Non-covered transition	Created-Refused
7	Manage customer relations, Customer	Coverage	Non-covered transition	Potential-Disqualified
8	Claim handling, Claim	Coverage	Non-covered initial state	Registered
9	Launch product campaign, Campaign	Coverage	Non-conformant transition	terminate, Active-Terminated
10	Claim handling, Payment	Coverage	Non-covered final state	Refused
11	Manage customer relations, Customer	Conformance	Non-conformant last state	update customer status, Dormant
12	Claim handling, Payment	Coverage	Non-covered transition	Authorized-Stopped
13	Claim handling, Claim	Coverage	Non-covered transition	Settled-Closed
14	Launch product campaign, Campaign	Coverage	Non-conformant last state	terminate, Terminated
15	Claim handling, Claim	Conformance	Non-conformant transition	evaluate claim, Opened-Rejected
16	Claim handling, Claim	Conformance	Non-conformant transition	evaluate claim, Quoted-Applied
17	Develop campaign, Campaign	Conformance	Non-conformant first state	register, Initiated
18	Claim handling, Payment	Coverage	Non-covered final state	Stopped
19	Claim handling, Claim	Conformance	Non-conformant transition	evaluate claim, Opened-Granted

(b)



(c)

	No.	Model context	Inconsistency category	Inconsistency type	Model element context
p1	1	**Claim handling**, Claim	Conformance	Non-conformant transition	**register**, Received-Opened
	2	**Claim handling**, Payment	Conformance	Non-conformant transition	**carry out settlement**, Created-Paid in full
	3	**Claim handling**, Claim	Conformance	Non-conformant transition	**close claim**, Paid-Closed
	4	Claim handling, **Payment**	Coverage	Non-covered transition	**Created-Refused**
	5	Claim handling, **Claim**	Coverage	Non-covered initial state	**Registered**
	6	Claim handling, **Payment**	Coverage	Non-covered final state	**Refused**
	7	Claim handling, **Payment**	Coverage	Non-covered transition	**Authorized-Stopped**
	8	Claim handling, **Claim**	Coverage	Non-covered transition	**Settled-Closed**
	9	**Claim handling**, Claim	Conformance	Non-conformant transition	**evaluate claim**, Opened-Rejected
	10	**Claim handling**, Policy	Conformance	Non-conformant transition	**evaluate claim**, Quoted-Applied
	11	Claim handling, **Payment**	Coverage	Non-covered final state	**Stopped**
	12	**Claim handling**, Claim	Conformance	Non-conformant transition	**evaluate claim**, Opened-Granted
p2	13	Close account, Account	Conformance	Non-conformant transition	close, Opened-Closed
	14	Close account, Customer	Conformance	Non-conformant transition	update customer status, Active-Dormant
	15	Manage customer relations, Cu	Coverage	Non-covered transition	Potential-Disqualified
	16	Manage customer relations, Cu	Conformance	Non-conformant last state	update customer status, Dormant
p3	17	Launch product campaign, Car	Coverage	Non-conformant transition	terminate, Active-Terminated
	18	Launch product campaign, Car	Coverage	Non-conformant last state	terminate, Terminated
	19	Develop campaign, Campaign	Conformance	Non-conformant first state	register, Initiated

Figure 5.6: (a) Detected inconsistencies (b) Model partitions (c) Inconsistencies grouped by model partition

graphical representation. The models containing elements in the abstract model element context of an inconsistency form its abstract model context.

Definition 61 (Abstract model element context, Abstract model context)**.** *Given an inconsistency* $i = it(me_j, ..., me_k)$ *in a model* M*, its* abstract model element context*,* $amec(i) \subseteq \{me_j, ..., me_k\}$*, is a non-empty subset of its model element context. The* abstract model context *of inconsistency* i*,* $amc(i)$*, is a set of submodels in* M *where each submodel contains at least one element in* $amec(i)$*.*

For process and object life cycle inconsistencies, we can define abstract model element contexts as follows:

- $ncnf\_tran(a, s_1, s_2)$, $ncnf\_init(a, s_{first})$, $ncnf\_fin(n, s_{last})$: action a;
- $ncov\_tran(s_1, s_2)$: transition from s_1 to s_2;
- $ncov\_init(s_i)$, $ncov\_fin(s_f)$: states s_i and s_f respectively.

Assuming these abstract model element contexts, the abstract model context for non-conformance inconsistencies is always a process model, while for non-coverage inconsistencies it is an object life cycle model. For example, in the model partition $p1$ shown in Figure 5.6(c), elements of abstract model element contexts and abstract model contexts are marked in bold in the Model element context and Model context columns, respectively.

We now define two types of context-switches that occur during resolution of inconsistencies: *model element change* and *model change*.

Definition 62 (Model element change, Model change)**.** *Moving between inconsistencies* i_1 *and* i_2 *requires a* model element change *if the abstract model element contexts of* i_1 *and* i_2 *do not overlap, i.e.* $amec(i_1) \cap amec(i_2) = \emptyset$*. Moving between* i_1 *and* i_2 *requires a* model change *if the abstract model contexts of* i_1 *and* i_2 *do not overlap, i.e.* $amc(i_1) \cap amc(i_2) = \emptyset$*.*

In Figure 5.6(c), moving from inconsistency 1 to inconsistency 2 requires a model element change, but no model change. On the other hand, no model element change is required when moving from inconsistency 9 to 10.

We aim not only to reduce the number of model element changes and model changes that the modeler has to perform during inconsistency resolution, but also to avoid taking the modeler "back and forth" in the same model while moving between inconsistencies or reduce the number of so-called *backward moves*. A partial order $\leq_{amec}$ needs to be defined on abstract element contexts of inconsistencies using some inherent relations between these elements in their respective model contexts.

Abstract model element contexts of non-conformance inconsistencies always comprise actions in a process model. Since these are already organized into a directed graph for the same process model, it is easy to define the partial order $\leq_{amec}$. Actions in different process models simply remain unordered. For non-coverage inconsistencies, some abstract model element contexts comprise transitions and others comprise states in an object life cycle model. In this case, a partial order $\leq_{amec}$ can be defined by comparing the distances from the initial state to the source states of transitions or to states themselves.

Definition 63 (Backward move)**.** *Let a model* M *and its associated set of inconsistencies* $I_M = \{i_1, ..., i_n\}$ *be given. Assuming that a partial order relation* $\leq_{amec}$ *is defined on the set of all abstract model element contexts* $\{amec(i_1), ..., amec(i_n)\}$*, a model element change from* i_j *to* i_k *is called a* backward move *in one of the following two cases:*

- *if no model change is required from i_j to i_k and $amec(i_k) \leq_{amec} amec(i_j)$;*
- *if a model change is required from i_j to i_k and $amec(i_k) \leq_{amec} amec(i_x)$, where i_x is the last inconsistency processed in $amc(i_k)$.*

For example, let us consider the modeler resolving the inconsistencies in model partition $p1$ in the order shown in Figure 5.6(c). This would require 8 model changes and 11 model element changes, one of which is a backward move. Figure 5.7 shows the details of these context-switches. The No. column shows at which inconsistency the change would take place and Target AMC and Target AMEC show to what abstract model and model element contexts the modeler would change, respectively.

Model element changes

No.	Target AMEC
1	register
2	carry out settlement
3	close claim
4	Created-Refused
5	Registered
6	Refused
7	Authorized-Stopped
8	Settled-Closed
9	**evaluate claim**
11	Stopped
12	evaluate claim

Model changes

No.	Target AMC
1	Claim handling
4	Payment
5	Claim
6	Payment
8	Claim
9	Claim handling
11	Payment
12	Claim handling

Figure 5.7: Example context-switches

In Figure 5.7, the model element change to inconsistency 9 is a backward move, since it requires a model change and $evaluate\ claim$ action $\leq_{amec}$ $close\ claim$ action, which was the last processed abstract model element context for the $Claims\ handling$ model context (assuming that $close\ claim$ appears right at the end of this process model).

We have now introduced all the main concepts necessary to perform the inconsistency prioritization illustrated in Figure 5.5. To take into account modeler-defined priorities, which may express domain-specific concerns, we assume the modeler can assign priorities to resolving inconsistencies in different models and to resolving inconsistencies of different types. The inconsistency prioritization is then performed as follows:

1. group inconsistencies by model partitions to produce groups $A1, ..., Am$;
2. order groups $A1, ..., Am$ by average model priority in the model partitions;
3. group inconsistencies inside each group Ai by abstract model context to produce groups $B1, ..., Bn$;
4. order groups $B1, ..., Bn$ by average model priority;
5. group inconsistencies inside each group Bj by abstract model element context to produce groups $C1, ..., Cp$;
6. order groups $C1, ..., Cp$ by $\leq_{amec}$ partial order;
7. group inconsistencies inside each group Ck by inconsistency type to produce groups $D1, ..., Dq$;
8. order groups $D1, ..., Dq$ by inconsistency type priority.

During inconsistency resolution according to the final prioritized set of inconsistencies, context-switches are minimized as follows. Model changes and model element changes are reduced due to the grouping of inconsistencies by their abstract model contexts and abstract model element contexts, since when all inconsistencies with the same abstract

context appear together in the list, this context only needs to be visited once. Backward moves are avoided, because the inconsistencies are ordered according the partial order $\leq_{amec}$ defined for their abstract model element contexts.

Model partitions do not have a direct effect on the minimization of context-switches, but they facilitate the identification of inconsistency groups that can be handled independently from each other. Priorities of inconsistency types are used in the ordering to consider modeler-defined priorities.

Going back to the example shown in Figure 5.6, we now demonstrate the application of the inconsistency prioritization to the inconsistencies in the model partition $p1$ in Figure 5.6(c). The resulting prioritized inconsistency list is shown in Figure 5.8. Abstract model contexts and abstract model element contexts are shown in bold in the Model context and Model element context columns, respectively. This list shows that the modeler has assigned the highest priority to resolving inconsistencies in the $Claims\ handling$ process model and that the priority of $Payment$ object life cycle model is higher than that for the $Claim$ object life cycle model. Resolving inconsistencies in this order would require the modeler to make 3 model changes and 10 model element changes without any backward moves (in contrast to the original 8 model changes and 11 model element changes with 1 backward move).

		Model context	Model element context	Inconsistency type	Inconsistency category
p1	1	**Claim handling**, Claim	**register**, Received-Opened	Non-conformant transition	Conformance
	2	**Claim handling**, Payment	**carry out settlement**, Created-Paid in full	Non-conformant transition	Conformance
	3	**Claim handling**, Policy	**evaluate claim**, Quoted-Applied	Non-conformant transition	Conformance
	4	**Claim handling**, Claim	**evaluate claim**, Opened-Rejected	Non-conformant transition	Conformance
	5	**Claim handling**, Claim	**evaluate claim**, Opened-Granted	Non-conformant transition	Conformance
	6	**Claim handling**, Claim	**close claim**, Paid-Closed	Non-conformant transition	Conformance
	7	Claim handling, **Payment**	**Created-Refused**	Non-covered transition	Coverage
	8	Claim handling, **Payment**	**Authorized-Stopped**	Non-covered transition	Coverage
	9	Claim handling, **Payment**	**Refused**	Non-covered final state	Coverage
	10	Claim handling, **Payment**	**Stopped**	Non-covered final state	Coverage
	11	Claim handling, **Claim**	**Registered**	Non-covered initial state	Coverage
	12	Claim handling, **Claim**	**Settled-Closed**	Non-covered transition	Coverage

Figure 5.8: Grouped and ordered inconsistencies

In this section, we have presented our approach to inconsistency prioritization. A set of detected inconsistencies can now be automatically prioritized according to this approach, after which the modeler can systematically move through the inconsistencies in the determined order. In the following section, we describe how side-effect forecast can assist the modeler in analyzing the impact of resolutions for one particular inconsistency (cf. Section 5.2.2).

5.4 Side-Effect Forecast for Impact Analysis of Resolutions

In this section, we describe how resolution side-effects can be forecasted based on the specification of so-called side-effect expressions by the tool developer during the development of resolution types. This section and the following section on cost-based resolution comparison are based on our earlier publication [Küster and Ryndina, 2007].

In view of developing tool support for inconsistency resolution, the main task of the tool developer is to define resolution types and to develop model transformations for each resolution type. Examples of informal specifications of such model transformations were shown in Figure 5.2. To facilitate the forecasting of resolution side-effects, we suggest that the tool developer additionally makes the dependencies between each resolution

type and inconsistency type explicit. These can be captured directly as side-effect expressions of a resolution type. A side-effect expression defines how a set of induced or expired inconsistencies of a particular type can be computed based on a given inconsistency.

Definition 64 (Side-effect and side-effect expression)**.** *Let a model M and an inconsistency type $it \in \mathcal{I}_M$ be given. A resolution type $rt \in \mathcal{R}_{it}$ is associated with* side-effects *$E_{rt} = {E_{rt}}^- \cup {E_{rt}}^+$, where ${E_{rt}}^-$ comprise expired inconsistencies and ${E_{rt}}^+$ comprise induced inconsistencies. Each side-effect comprises inconsistencies of one type, defined by function $type : E_{rt} \rightarrow \mathcal{I}_M$. A side-effect $e \in E_{rt}$ is associated with a* side-effect expression *$exp_e : I_M \rightarrow \mathcal{P}(I_{M'})$, where M' denotes the model that would be obtained by applying a resolution $rt(me_j, ..., me_k)$ to an inconsistency $i = it(me_j, ..., me_k) \in I_M$.*

Let us consider an example of side-effect expressions for resolution types defined for process and object life cycle inconsistencies. Figure 5.9 shows the claims handling process model and the object life cycle model for claims, introduced in Section 4.3 of Chapter 4. The following inconsistencies exist between these models:

- $ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$;
- $ncnf\_tran(Settle\ Claim, Reviewed, Settled)$;
- $ncnf\_tran(Settle\ Claim, Reviewed, NeedsReview)$;
- $ncnf\_tran(Review, NeedsReview, Reviewed)$;
- $ncnf\_fin(EvaluateClaim, Rejected)$;
- $ncov\_tran(Rejected, Closed)$.

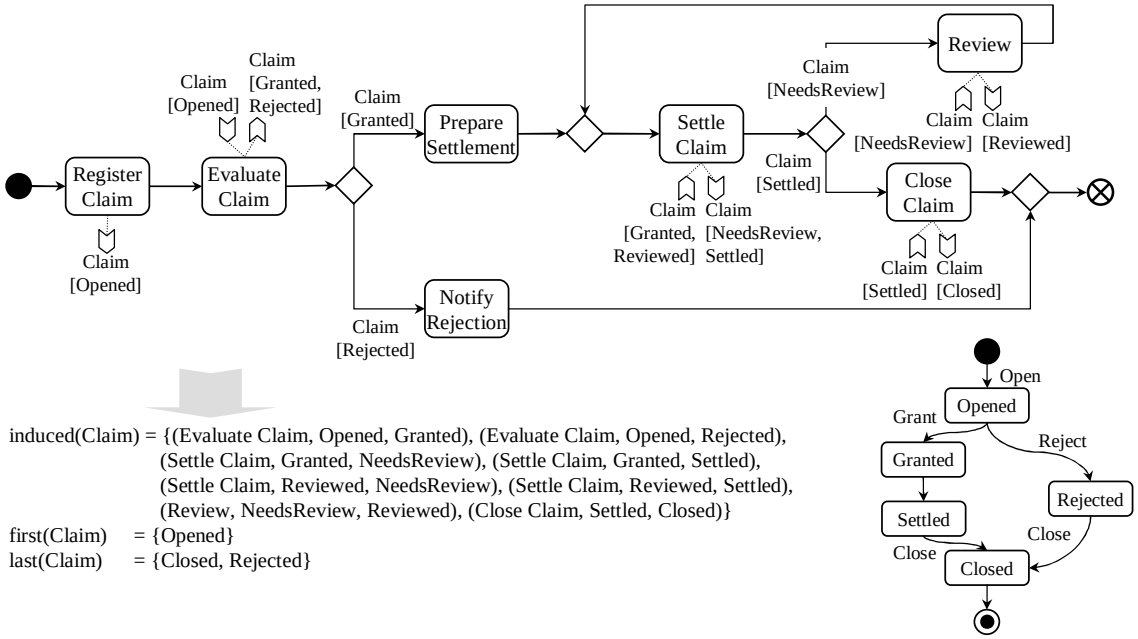


Figure 5.9: Inconsistencies example

Based on the resolution types rt_1, rt_2 and rt_3 shown in Figure 5.2, each of the inconsistencies $ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$ and $ncnf\_tran(Review,$

$NeedsReview$, $Reviewed$) can be resolved using three alternative resolutions, as shown in Figure 5.10. Some of these resolutions have side-effects. For example, applying r_1 to $ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$ as shown in Figure 5.10(a) introduces an induced inconsistency $ncov\_tran(Granted, Settled)$. Applying r_1 to $ncnf\_tran(Review,$ $NeedsReview, Reviewed)$ as shown in Figure 5.10(d) introduces an induced inconsistency $ncnf\_init(Reviewed)$. This example shows that applying resolutions of the same type to different inconsistencies may yield different side-effects.

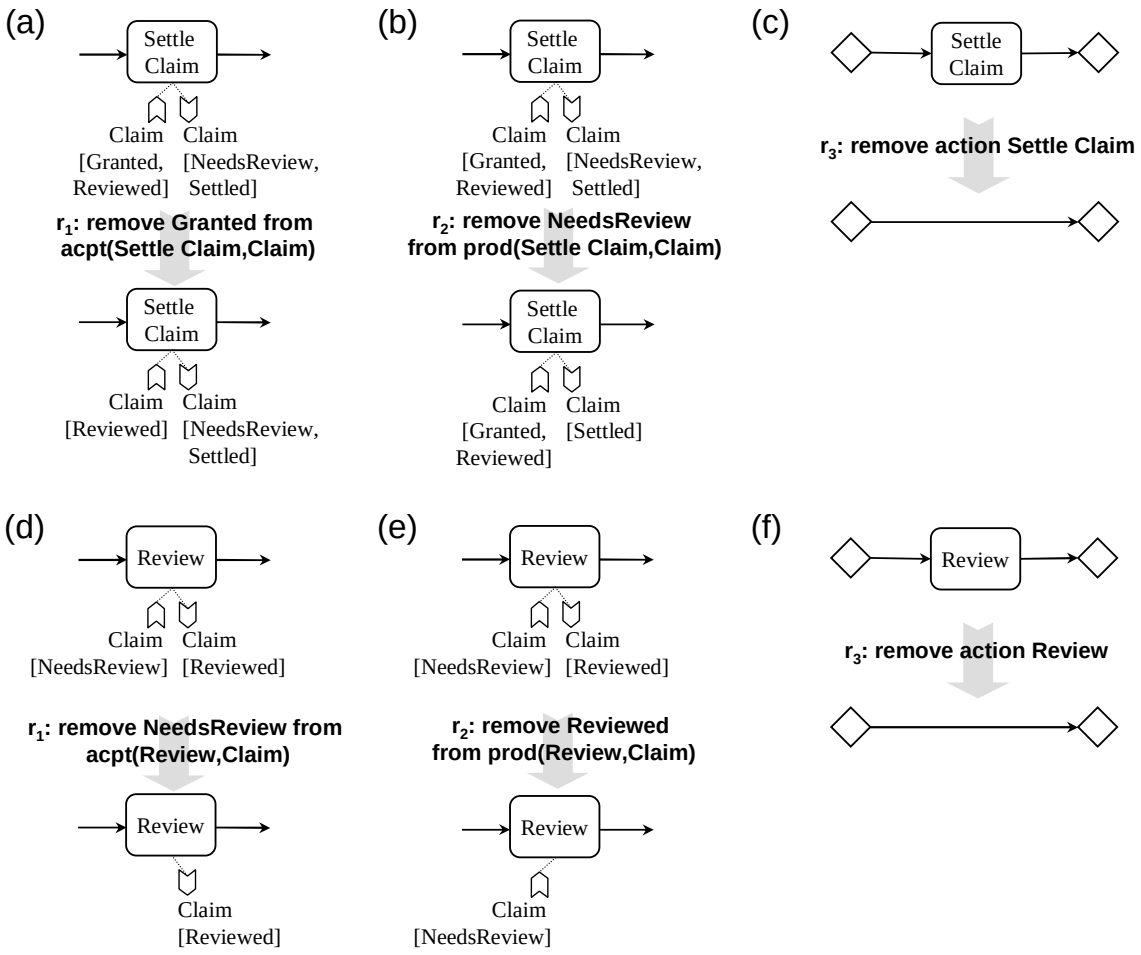


Figure 5.10: Resolving non-conformant transitions in the example

We now specify a side-effect expression for the resolution type rt_1 such that the side-effects of applying a resolution of this type to a given inconsistency can be forecasted. Applying a resolution of type rt_1 to a non-conformant transition (a, s_1, s_2) involves removing s_1 from $acpt(a, t)$, which may resolve non-conformant transitions that are induced by action a other than the target non-conformant transition. This means that rt_1 can expire existing non-conformant transitions, which we define in side-effect expression exp_{e_1} as follows:

$exp_{e_1}(ncnf\_tran(a, s_1, s_2)) = \{ncnf\_tran(a, s_1, s_l) \mid s_l \neq s_2$ and $s_l \in prod(a, t)$ and $ncnf\_tran(a, s_1, s_l) \in I_M\}$

According to this side-effect expression, an inconsistency $ncnf\_tran(a, s_1, s_l) \in I_M$ is expired as a result of applying the resolution $rt_1(a, s_1, s_2)$ if $s_l \neq s_2$ and $s_l \in prod(a, t)$.

In the example resolution shown in Figure 5.10(a), $a = Settle\ Claim$, $s_1 = Granted$ and $s_2 = NeedsReview$. In this case, there is no $ncnf\_tran(Settle\ Claim, Granted, s_l)$ such that $s_l \neq NeedsReview$. Therefore, no non-conformant transition inconsistencies are expired by applying this resolution.

In Table 5.1, we define side-effect expressions for resolution types rt_1, rt_2 and rt_3. We specify expressions semi-formally, although this could also be done using first-order logic or Object Constraint Language (OCL) [OCL, 2003].

Table 5.1: Side-effect expressions, where $i = ncnf\_tran(a, s_1, s_2)$ is an inconsistency between a workflow graph G and an object life cycle model OLC_t

Res	Side-effect	Side-effect expression
rt_1	$e_1 \in {E_{rt_1}}^-$	$exp_{e_1}(i) = \{ncnf\_tran(a, s_1, s_l) \mid s_l \neq s_2$ and $s_l \in prod(a,t)$ and $ncnf\_tran(a, s_1, s_l) \in I_M\}$
	$e_2 \in {E_{rt_1}}^-$	$exp_{e_2}(i) = \{ncov\_init(s_l) \mid s_l \in prod(a,t)$ and s_1 is the only state in $acpt(a,t)$ and $ncov\_init(s_l) \in I_M\}$
	$e_3 \in {E_{rt_1}}^+$	$exp_{e_3}(i) = \{ncov\_tran(s_1, s_l) \mid (a, s_1, s_l)$ is an induced transition of t in G and the only element in the coverage set of $(s_1, s_l)\}$
	$e_4 \in {E_{rt_1}}^+$	$exp_{e_4}(i) = \{ncnf\_init(a, s_l) \mid s_l \in prod(a,t)$ and s_1 is the only state in $acpt(a,t)$ and there is no transition from the initial state to s_l in $OLC_t\}$
rt_2	$e_5 \in {E_{rt_1}}^-$	$exp_{e_5}(i) = \{ncnf\_tran(a, s_k, s_2) \mid s_k \neq s_1$ and $s_k \in acpt(a,t)$ and $ncnf\_tran(a, s_k, s_2) \in I_M\}$
	$e_6 \in {E_{rt_2}}^-$	$exp_{e_6}(i) = \{ncnf\_fin(a, s_2) \mid ncnf\_fin(a, s_2) \in I_M\}$
	$e_7 \in {E_{rt_2}}^-$	$exp_{e_7}(i) = \{ncov\_fin(s_k) \mid s_k \in effin(a,t)$ and s_2 is a last state of t in G and the only state in $prod(a,t)$ and $ncov\_fin(s_k) \in I_M\}$
	$e_8 \in {E_{rt_2}}^+$	$exp_{e_8}(i) = \{ncov\_tran(s_k, s_2) \mid (a, s_k, s_2)$ is an induced transition of t in G and (a, s_k, s_2) is the only element in the coverage set of $(s_k, s_2)\}$
	$e_9 \in {E_{rt_2}}^+$	$exp_{e_9}(i) = \{ncov\_fin(s_2) \mid (a, s_2)$ is the only element in the coverage set of the final transition to $s_2\}$
	$e_{10} \in {E_{rt_2}}^+$	$exp_{e_{10}}(i) = \{ncnf\_fin(a, s_k) \mid s_k \in effin(a,t)$ and s_2 is a last state of t in G and the only state in $prod(a,t)$ and there is no final transition from s_k in $OLC_t\}$
rt_3	$e_{11} \in {E_{rt_3}}^-$	$exp_{e_{11}}(i) = \{ncnf\_tran(a, s_k, s_l) \mid s_k \in acpt(a,t)$ and $s_l \in prod(a,t)$ and $ncnf\_tran(a, s_k, s_l) \in I_M\}$
	$e_{12} \in {E_{rt_3}}^-$	$exp_{e_{12}}(i) = \{ncnf\_fin(a, s_l) \mid s_l \in prod(a,t)$ and $ncnf\_fin(a, s_l) \in I_M\}$
	$e_{13} \in {E_{rt_3}}^-$	$exp_{e_{13}}(i) = \{ncov\_fin(s_k) \mid s_k \in effin(a,t)$ and s_2 is a last state of t in G and $ncov\_fin(s_k) \in I_M\}$
	$e_{14} \in {E_{rt_3}}^+$	$exp_{e_{14}}(i) = \{ncov\_tran(s_k, s_l) \mid (a, s_k, s_l)$ is an induced transition of t in G and the only element in the coverage set of $(s_k, s_l)\}$
	$e_{15} \in {E_{rt_3}}^+$	$exp_{e_{15}}(i) = \{ncov\_fin(s_l) \mid s_l \in prod(a,t)$ and (a, s_l) is the only element in the coverage set of the final transition to $s_l\}$
	$e_{16} \in {E_{rt_3}}^+$	$exp_{e_{16}}(i) = \{ncnf\_fin(n_1, s_k) \mid n_1$ is the predecessor node of a and $s_k \in effin(a,t)$ and s_2 is a last state of t in G and there is no final transition from s_k in OLC_t

Some induced transitions that provide coverage for a transition in a given object life cycle model may no longer be induced after a resolution of type rt_1 is applied. To capture this in a side-effect expression, we introduce the concept of a coverage set: A *coverage set* of a transition (s_k, s_l) in a given object life cycle model contains all induced transitions of the form (a, s_k, s_l) in a given process model. If an induced transition (a, s_k, s_l) is the only

element of a coverage set for (s_k, s_l), then removing this induced transition will introduce a new non-covered transition (s_k, s_l). For rt_1, the concept of a coverage set is used to define exp_{e_3}. The concept of a coverage set also applies to initial and final transitions.

Let us now once again consider resolving inconsistencies $ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$ and $ncnf\_tran(Review, NeedsReview, Reviewed)$ in our example using the resolutions shown in Figure 5.10. We get the following details about the impact of these resolutions on the overall set of inconsistencies in the underlying models if we evaluate the defined side-effect expressions:

Resolutions for $ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$:

r_1 resolves $ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$ and introduces $ncov\_tran(Granted, Settled)$;

r_2 resolves $ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$, $ncnf\_tran(Settle\ Claim, Reviewed, NeedsReview)$;

r_3 resolves $ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$, $ncnf\_tran(Settle\ Claim, Reviewed, NeedsReview)$, $ncnf\_tran(Settle\ Claim, Reviewed, Settled)$ and introduces $ncov\_tran(Granted, Settled)$.

Resolutions for $ncnf\_tran(Review, NeedsReview, Reviewed)$:

r_1 resolves $ncnf\_tran(Review, NeedsReview, Reviewed)$ and introduces $ncnf\_init(Review, Reviewed)$;

r_2 resolves $ncnf\_tran(Review, NeedsReview, Reviewed)$;

r_3 resolves $ncnf\_tran(Review, NeedsReview, Reviewed)$.

Such additional information assists the modeler in the overall impact analysis of each resolution available for a particular inconsistency. The modeler is relieved from manually examining the models and the current inconsistency set to determine the side-effects of each resolution. This additional benefit can of course only be provided if the tool developer contributes by specifying and implementing the side-effect expressions. However, the side-effect expressions only need to be specified once during the tool development, after which the modeler can repeatedly reap the resulting benefits.

The forecasted side-effects can also be leveraged in the comparison of alternative resolutions (cf. Section 5.2.2), as we explain in the next section.

5.5 Cost-Based Comparison of Alternative Resolutions

Based on the side-effect forecast, the modeler is presented with the detailed information about the overall impact of each resolution. The modeler can directly use this information to compare alternative resolutions available for the same inconsistency. Generally, the most beneficial resolution would be the one that overall removes the greatest number of inconsistencies.

The *effect* of a resolution can be quantified as the overall effect its application has on the total number of inconsistencies in the given model.

Definition 65 (Effect)**.** *Given a resolution $r = rt(me_j, ..., me_k)$ that resolves an inconsistency $i = it(me_j, ..., me_k) \in I_M$ of type $it \in \mathcal{I}_M$, with rt side-effects $E^-_{rt} = \{e_{11}, ..., e_{1p}\}$ and $E^+_{rt} = \{e_{21}, ..., e_{2q}\}$, the* effect *of resolution r is denoted by $effect_r$ and calculated as follows:*

$$\mathit{effect}_r = \sum_{k=1}^{q} (\mid exp_{e_{2k}}(i) \mid) - \sum_{j=1}^{p} \big(\mid exp_{e_{1j}}(i) \mid\big) - 1$$

In the example shown in Figure 5.9, the effects of resolutions r_1, r_2, r_3 in Figure 5.10 are as follows:

$ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$:
$\mathit{effect}_{r_1} = 1 - 1 = 0$
$\mathit{effect}_{r_2} = -1 - 1 = -2$
$\mathit{effect}_{r_3} = 1 - 2 - 1 = -2$

$ncnf\_tran(Review, NeedsReview, Review)$:
$\mathit{effect}_{r_1} = 1 - 1 = 0$
$\mathit{effect}_{r_2} = -1$
$\mathit{effect}_{r_3} = -1$

According to these effect values of the resolutions, we would choose r_2 or r_3 to resolve $ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$ and $ncnf\_tran(Review, NeedsReview, Reviewed)$.

As part of the inconsistency prioritization, we have already discussed that resolution of inconsistencies of different types may have different priority. Comparing alternative resolution based only on their effects would ignore such relative priorities. A more fine-grained comparison of alternative resolutions can be facilitated by taking into account the different priorities of inconsistencies. We hence introduce *costs* to reflect relative severity of different inconsistency types and to quantify the total inconsistency cost of a given model.

Definition 66 (Inconsistency type cost and total inconsistency cost)**.** *Let a model M, its associated inconsistency types $\mathcal{I}_M$ and the set of inconsistencies I_M in M be given.* Inconsistency type cost *is defined by a function $cost : \mathcal{I}_M \rightarrow \mathbb{N}$ that maps an inconsistency type $it \in \mathcal{I}_M$ to a natural number. The* total inconsistency cost *for M is the sum of the inconsistency type costs for all inconsistencies in M: $\sum_{it(me_j,...,me_k) \in I_M} cost(it)$.*

Costs can either be assigned to inconsistency types once and then used for inconsistency resolution in every model, or different costs can be assigned for each model to reflect a specific resolution goal. For our example, we assume that our main goal is to achieve object life cycle conformance of the claims handling process model. We further consider that conformance of transitions and final transitions is more important than that of initial transitions. Therefore, we assign the following costs to the different inconsistency types: $cost(ncnf\_tran) = 3$, $cost(ncnf\_init) = 2$, $cost(ncnf\_fin) = 3$, $cost(ncov\_tran) = 1$, $cost(ncov\_init) = 1$ and $cost(ncov\_fin) = 1$.

Inconsistency type costs facilitate a more fine-grained comparison of resolutions based on *cost reduction* values calculated for each resolution.

Definition 67 (Cost reduction)**.** *Given a resolution $r = rt(me_j, ..., me_k)$ that resolves an inconsistency $i = it(me_j, ..., me_k) \in I_M$ of type $it \in \mathcal{I}_M$, with rt side-effects $E^-_{rt} = \{e_{11}, ..., e_{1p}\}$ and $E^+_{rt} = \{e_{21}, ..., e_{2q}\}$, the* cost reduction *$r$ is denoted by $costred_r$ and calculated as follows:*

$$costred_r = cost(it) + \sum_{j=1}^{p} \big(\mid exp_{e_{1j}}(i) \mid \times cost(type(e_{1j}))\big) - \sum_{k=1}^{q} (\mid exp_{e_{2k}}(i) \mid \times cost(type(e_{2k})))$$

We now calculate cost reduction values for the resolutions in our example:

$ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$:
$costred_{r_1} = 3 - (1 \times 1) = 2$
$costred_{r_2} = 3 + (1 \times 3) = 6$
$costred_{r_3} = 3 + (2 \times 3) - (1 \times 1) = 8$

$ncnf\_tran(Review, NeedsReview, Reviewed)$:
$costred_{r_1} = 3 - (1 \times 2) = 1$
$costred_{r_2} = 3$
$costred_{r_3} = 3$

It can be seen that based on the calculated cost reduction values, we can perform a more fine-grained comparison of the resolutions that takes into account the priorities or costs of different inconsistency types. While both resolutions r_2 and r_3 reduce the total number of inconsistencies by 2 when applied to the $ncnf\_tran(Settle\ Claim, Granted, NeedsReview)$ inconsistency, the cost reduction values show that r_3 is more beneficial. It resolves 3 inconsistencies of the most severe type, non-conformant transition, and introduces an inconsistency of low severity, a non-covered transition.

With our approach, we could also introduce more automation into the resolution process by applying resolutions without the intervention of the modeler whenever there is one resolution that has a highest cost reduction value for a particular inconsistency. However, in our scenario, approving the choice of a resolution needs to be done by an expert who is aware of what impact the change in the process model has on the business. Provided that we are working with a model of an existing business process, removing an action from this model translates to removing a task in the process and may be difficult to implement in practice, even though the cost reduction value indicates that this is the best resolution.

Even if a particular resolution is ranked best according to its effect and cost reduction, it may not be desirable to apply it if it can lead to a resolution cycle. In the following, we describe an approach to addressing this challenge (cf. Section 5.2.4).

5.6 Assignment of Cycle Safety Categories to Resolutions

In this section, we describe how the information captured in side-effect expressions of resolution types can be used for detection of resolution cycles and assignment of so-called *cycle safety categories* to resolutions. We propose an analysis that combines instance-level information with type-level information. At the instance level, we directly consider a particular inconsistency, its resolutions and the forecasted side-effects of these resolutions in searching for resolution cycles. On the type level, we take into account dependencies between inconsistency and resolution types. A combination of analyses on these two levels allows us to detect cases where a resolution leads to a resolution cycle.

In the following description of our approach, we use the following definition of a resolution cycle.

Definition 68 (Resolution cycle)**.** *A* resolution cycle *is a sequence of tuples* $(i_1, r_1), ..., (i_m, r_m)$ *such that resolution* r_j *resolves inconsistency* i_j *for* $1 \leq j \leq m$*, resolution* r_j *introduces inconsistency* i_{j+1} *as a side-effect for* $1 \leq j \leq m-1$ *and resolution* r_m *introduces inconsistency* i_1 *as a side-effect.*

In order to determine whether a resolution can lead to a resolution cycle when applied to a concrete inconsistency, we analyze the *resolution tree* of this inconsistency.

Definition 69 (Resolution tree)**.** *A* resolution tree *of an inconsistency* i *is a tree that contains nodes labeled with inconsistencies and resolutions, and is constructed as follows:*

- *A root node labeled* i *is created.*

- *Given a node labeled with inconsistency i_j, a child node n is created for every resolution r_j that can resolve inconsistency i_j. Node n is labeled r_j.*
- *Given a node labeled with resolution r_k, a child node n is created for every inconsistency i_k that is introduced as a side-effect of r_k. Node n is labeled i_k.*

An example of a resolution cycle is shown in the resolution tree of inconsistency i_1 in Figure 5.11. It can be seen that an occurrence of a resolution cycle leads to an infinite resolution tree.

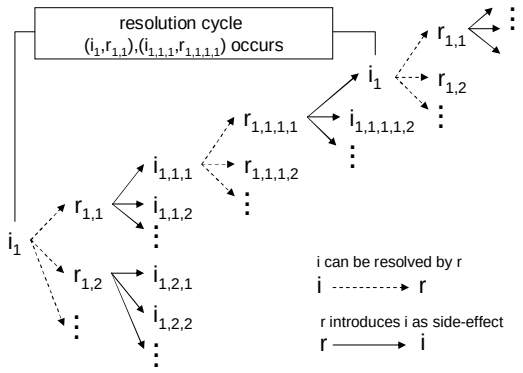


Figure 5.11: Resolution tree and example of resolution cycle

Given a resolution r that resolves an inconsistency i, we analyze whether beginning with this resolution it is possible to resolve all the inconsistencies subsequently introduced as side-effects without a resolution cycle occurring. We introduce the concept of a *resolution strategy tree* to explain this:

Definition 70 (Resolution strategy tree)**.** *Given an inconsistency i and a resolution r that resolves i, a* resolution strategy tree *for (i, r) is a tree that contains nodes labeled with tuples (i_j, r_j) and is constructed as follows:*

- *A root node labeled (i, r) is created.*
- *Given a node labeled (i_j, r_j), a child node n is created for every inconsistency i_k that is introduced as a side-effect of r_j. Node n is labeled (i_k, r_k) where r_k is one of the possible resolutions for i_k.*

A preorder traversal of a resolution strategy tree for (i, r) represents one possible resolution sequence or a *resolution strategy* beginning with resolution r that resolves i and all inconsistencies subsequently introduced as side-effects. There may be more than one resolution strategy tree for the same tuple (i, r) and there may also be more than one resolution strategy for the same resolution strategy tree. Given a tuple (i, r), all the possible resolution strategy trees can be constructed from the resolution tree of i.

Figure 5.12 shows two resolution strategy trees for the same tuple (i_1, r_1). Examples of resolution strategies for the resolution strategy tree in Figure 5.12(a) are: (i_1, r_1), (i_2, r_3), (i_5, r_7), (i_3, r_5), (i_7, r_{10}) and (i_1, r_1), (i_3, r_5), (i_7, r_{10}), (i_2, r_3), (i_5, r_7).

Definition 71 (Resolution strategy tree leads to resolution cycle)**.** *Given a resolution strategy tree for (i, r), all resolution strategies derived by traversing this tree lead to a resolution cycle if the resolution strategy tree is infinite. In this case, we say that the* resolution strategy tree leads to a resolution cycle.

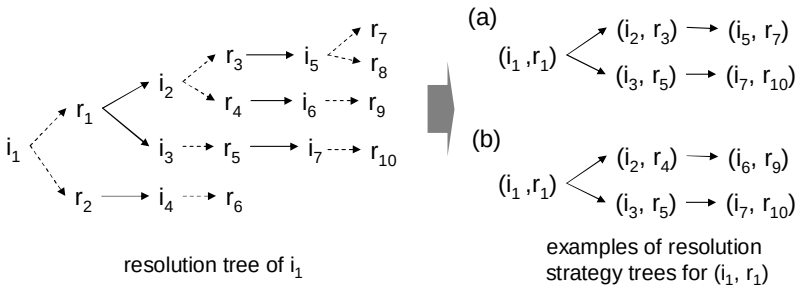


Figure 5.12: Examples of resolution strategy trees

Based on these definitions, we distinguish four cycle safety categories for resolutions:

Definition 72 (Cycle safety categories)**.** *A resolution r that resolves inconsistency i falls into one of the following* cycle safety categories*:*

- globally-safe*: none of the resolution strategy trees for (i, r) lead to a resolution cycle;*
- globally-unsafe*: every resolution strategy tree for (i, r) leads to a resolution cycle;*
- safe*: at least one resolution strategy tree for (i, r) does not lead to a resolution cycle;*
- safety-unknown*: it cannot be determined whether r falls into any of the other categories.*

5.6.1 Instance-Level Analysis

Resolution side-effect expressions allow us to forecast side-effects of resolutions, which we need to create a resolution tree and resolution strategy trees for a given inconsistency in a concrete model. In the following, we assume that applying a resolution r to inconsistency i does not affect the side-effects of other resolutions in the same resolution strategy tree that are not on the path from the root to the node labeled (i, r).

A complete safety analysis would search the entire resolution tree of an inconsistency, which can easily become intractable since the size of the tree grows exponentially with its depth. Therefore, we use a *lookahead* approach to our safety analysis to produce results in a reasonable time. Given an inconsistency in a concrete model and a lookahead value, a finite resolution tree can be constructed for this inconsistency. With a lookahead value x, each branch in this resolution tree would contain at most x resolutions and $x + 1$ inconsistencies. In Figure 5.11, given an inconsistency i_1 and a lookahead value of 2, the resolution tree would reach the depth of inconsistency $i_{1,1,1,1,2}$.

Once a resolution tree is constructed using a lookahead, all possible resolution strategy trees could be constructed from it and analyzed to determine a safety category for each resolution. We follow a simpler approach that works directly on the resolution tree and only implicitly uses the concept of a resolution strategy tree, as described in pseudocode in Listing 5.1.

In the pseudocode described in Listing 5.1, first the leaf nodes of the given resolution tree are assigned different cycle safety categories (lines 2-9). After that, the rest of the nodes in the tree are traversed, going from the leaf nodes to the root node and looking at all the nodes at a certain depth a time (lines 12-33). The safety category of a node is assigned based on the safety categories of its children nodes.

Listing 5.1: Resolution tree safety analysis

```
1  analyzeCycleSafety(Tree resolutionTree)
2    // determine cycle safety of leaf nodes
3    for each (Node leaf in resolutionTree.leaves) do
4      if (leaf is inconsistency that occurs more than once on path to root) then
5        leaf.category = globallyunsafe;
6      else if (leaf is inconsistency that occurs once on path to root) then
7        leaf.category = safetyunknown;
8      else if (leaf is resolution) then
9        leaf.category = globallysafe;
10   int depth = resolutionTree.depth - 1;
11   // determine cycle safety of remaining nodes
12   while (depth >= 0) do
13     for each (Node n in resolutionTree.nodesAtDepth(depth)) do
14       if (n is inconsistency) then
15         if (child.category = globallysafe for each child in n.childResolutions) then
16           n.category = globallysafe;
17         else if (child.category = globallyunsafe for each child in n.childResolutions) then
18           n.category = globallyunsafe;
19         else if (there exists child in n.childResolutions where child.category = globallysafe
20                  or child.category = safe) then
21           n.category = safe;
22         else
23           n.category = safetyunknown;
24       else if (n is resolution) then
25         if (child.category = globallysafe for each child in n.childInconsistencies) then
26           n.category = globally-safe;
27         else if (child.category = globallyunsafe for one child in n.childInconsistencies) then
28           n.category = globallyunsafe;
29         else if (child.category = safetyunknown for one child in n.childInconsistencies) then
30           n.category = safetyunknown;
31         else
32           n.category = safe;
33     depth--;
```

At the end of the described traversal of the given resolution tree, all resolutions in the resolution tree are placed into one of the safety categories. Resolutions are placed into the safety-unknown category when they occur on "non-cyclic" paths in the resolution tree that end with an inconsistency. For such resolutions, we apply a type-level safety analysis, which enables us to identify some further cases where such resolutions are globally-safe or safe.

5.6.2 Type-Level Analysis

A complete safety analysis on the instance level is expensive, while the lookahead approach is incomplete as it cannot determine whether some resolutions lead to cycles and places them into the safety-unknown category. In order to refine the results of the instance-level safety analysis with lookahead, we propose a type-level safety analysis that exploits the information about the types of inconsistencies that a resolution of a particular type can introduce as a side-effect.

Inconsistency types (e.g. non-conformant transition and non-covered initial transition) and examples of resolution types (e.g. remove accepted state and remove action) were introduced in Section 5.1. For each resolution type, a *resolution type tree* can be constructed showing the types of inconsistencies that it resolves and introduces as a side-effect. This basically abstracts the definition of a resolution tree to the type level. Figure 5.13 shows such trees for a sample set of resolution types, which we defined for resolving inconsistencies between process models with repository data flow and object life cycle models (not

detailed here).

For each inconsistency type it, resolution type trees are used to compute its so-called *side-effect sets*, where a side-effect set contains all the types of inconsistencies that can be introduced as side-effects under the same resolution strategy for an inconsistency of type it. The pseudocode in Listing 5.2 shows how one side-effect set for a given inconsistency type can be computed.

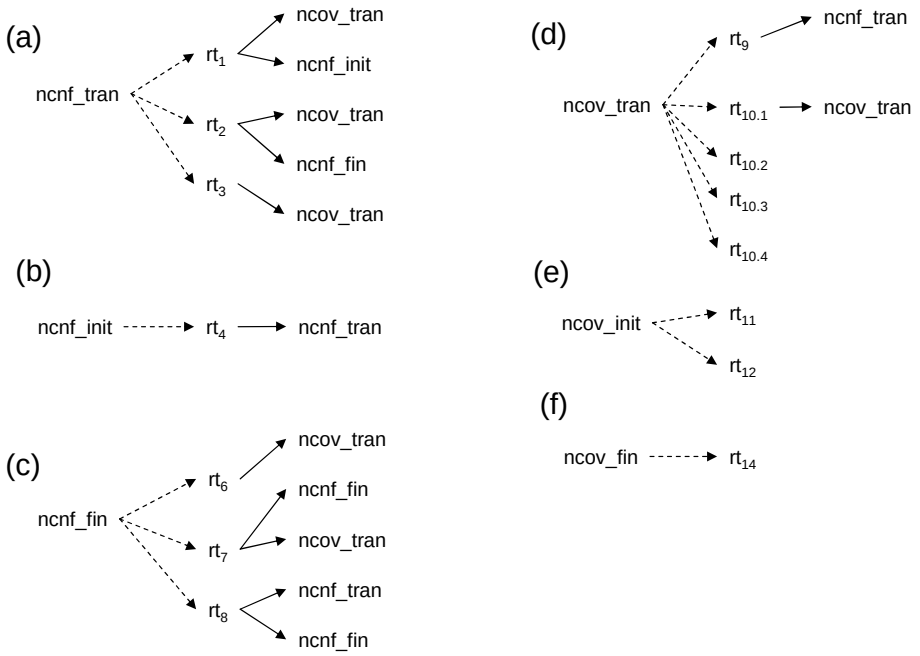


Figure 5.13: Resolution type trees

Listing 5.2: Side-effect set computation

```
1 Set getSideEffectSet(InconsistencyType it)
2   ResolutionType rt = it.oneAvailableResolutionType;
3   Set sideEffectSet = {};
4   Sequence strategy = (it, rt);
5   addSideEffects(rt, sideEffectSet, strategy);
6   return sideEffectSet
7
8 addSideEffects(ResolutionType rt, Set sideEffectSet, Sequence strategy)
9   if (rt.sideEffects has elements not contained in sideEffectSet) do
10    for each (InconsistencyType se in rt.sideEffects) do
11      sideEffectSet.add(se);
12      ResolutionType rt = se.oneAvailableResolutionType;
13      strategy.append((se, rt));
14      addSideEffects(rt, sideEffectSet, strategy);
```

Using the method outlined in Listing 5.2, all possible resolution strategies can be traversed and all side-effect sets for an inconsistency type determined. Since we always deal with a finite number of inconsistency and resolution types, all the side-effect sets are also finite. This computation only has to be done once when the tool supporting inconsistency resolution is implemented, after which side-effect sets for inconsistency types can be simply retrieved from storage when necessary.

For resolving an inconsistency of type $ncnf\_tran$, all resolution strategies that start with resolution type rt_1 give rise to the side-effect set $\{ncov\_tran, ncnf\_init, ncnf\_tran\}$, as

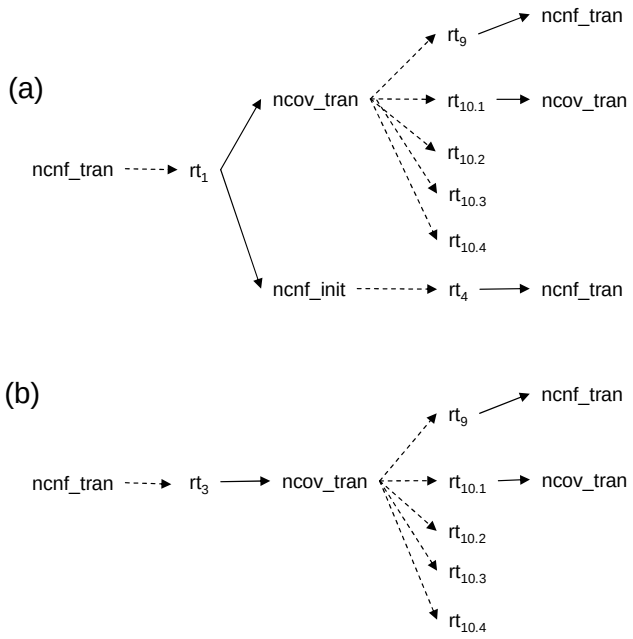


Figure 5.14: Analyzing resolution type trees

shown in Figure 5.14(a). On the other hand, resolution strategies starting with resolution type rt_3 give rise to $\{ncov\_tran, ncnf\_tran\}$ if they contain a resolution of type rt_9 and to $\{ncov\_tran\}$ in any other case, see Figure 5.14(b).

Given a resolution tree from the instance-level safety analysis and a resolution r that was placed into the safety-unknown category, we can use the pre-computed side-effect sets for further safety analysis of r as follows. First, leaf inconsistencies i_j marked safety-unknown are identified in the given tree. For each i_j, inconsistency types occurring on the path from the root to i_j in the resolution tree are determined. Then, side-effect sets of it_j, inconsistency type of i_j, are compared to the inconsistency types occurring on the path to i_j. If there are no matches found for at least one or all of the side-effect sets, then i_j is marked safe or globally-safe, respectively. Once all the leaves are checked in this way, the resolution tree safety analysis (Listing 5.1) is repeated. The algorithm in Listing 5.3 analyzes safety of leaf inconsistencies marked as safety-unknown.

Listing 5.3: Leaf inconsistency safety analysis

```
1  SafetyCategory getSafetyCategoryForLeafInconsistency ( Inconsistency i , ResolutionTree tree )
2    Path path = getPathFromRoot(tree, i);
3    Set inconsistencyTypesOnPath = getInconsistencyTypes(path);
4    it = i.inconsistencyType;
5    boolean globallySafe = true;
6    boolean safe = false;
7    for each (Set set in it.sideEffectSets ) do
8      if (set overlaps with inconsistencyTypesOnPath) then
9        globallySafe = false;
10     else
11       safe = true;
12   if ( globallySafe ) then
13     return globallysafe ;
14   else if (safe) then
15     return safe;
16   else
17     return safetyunknown;
```

After this refined safety analysis, which takes into account instance-level and type-level information, some safety-unknown resolutions will be categorized as globally-safe or safe. Those resolutions that remain in the safety-unknown category can potentially lead to a resolution cycle.

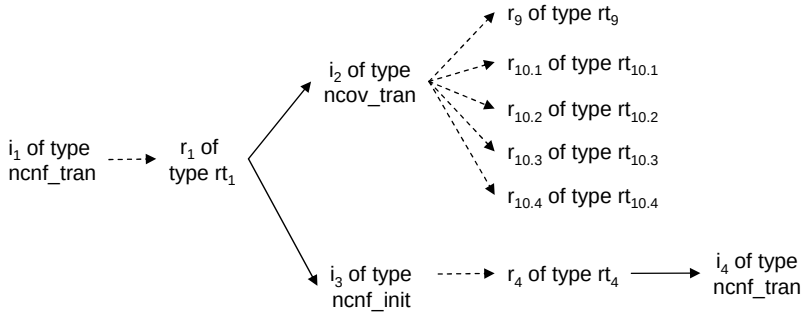


Figure 5.15: Extract from a resolution tree

Suppose that we are performing a safety analysis for resolution r_1 shown in Figure 5.15. Instance-level safety analysis with a lookahead of 2 places r_1 into the safety-unknown category since the inconsistency i_4 introduced as a side-effect of r_1 followed by r_4 cannot be resolved on the paths considered with lookahead of 2. Inconsistency i_4 is then put through a type-level safety analysis. The path to i_4 in the resolution tree contains inconsistencies of types $\{ncnf\_tran, ncnf\_init\}$. Inconsistency i_4 is of type $ncnf\_tran$, which among others has side-effect sets $\{ncov\_tran, ncnf\_tran\}$ and $\{ncov\_tran\}$, see Figure 5.14(b). Since one of these sets does not overlap with $\{ncnf\_tran, ncnf\_init\}$, inconsistency i_4 is marked as safe. Taking this new marking into consideration, another run through the resolution tree with the algorithm in Listing 5.1 determines that resolution r_1 is also safe.

Computation of resolution cycle safety assists the modeler in selecting among alternative resolutions for a particular inconsistency. In order to avoid resolution cycles, the modeler must always choose globally-safe or safe resolutions. In certain cases, no such resolution may be available or due to other requirements the modeler may need to choose a resolution in another category. Since at this point the modeler embarks on a potentially dangerous resolution path, a realization of our approach could allow the modeler to create a roll-back reference point at this time. If a cycle occurs at a later stage, the modeler can always roll-back and undo some resolutions.

With this, we conclude the presentation of our solution to the four challenges facing the modeler during inconsistency resolution, namely inconsistency prioritization and context-switching, impact analysis of resolutions, comparison of alternative resolutions and avoidance of resolution cycles. In the following section, we demonstrate how our solution can be embedded into the overall process for inconsistency management.

5.7 Augmenting the Inconsistency Management Process

In this section, we demonstrate our solution in the broader context of inconsistency management, which comprises activities other than just those related to the resolution of inconsistencies. The groping of inconsistency management activities into phases has been studied in the works of Finkelstein et al [Finkelstein et al., 1996] and Nuseibeh and Easterbrook [Nuseibeh and Easterbrook, 1999]. Spanoudakis and Zisman later collated these works in an extensive survey of the inconsistency management

field [Spanoudakis and Zisman, 2001]. In Figure 5.16, we depict the inconsistency management phases described in [Spanoudakis and Zisman, 2001].

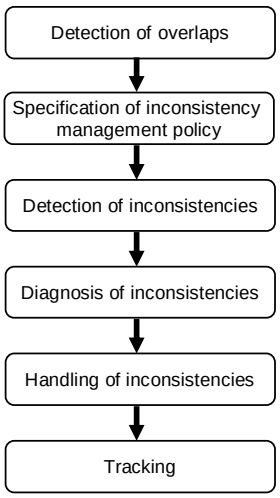


Figure 5.16: Inconsistency management process

Inconsistency management is dependent on a clear consistency definition, which is established during the detection of overlaps phase. In the terminology introduced in Chapter 4, this phase is concerned with the definition of a consistency concept. Since consistency concepts are defined in terms of the modeling languages at hand, the detection of overlaps is concerned with activities on the type level.

The specification of inconsistency management policy phase is concerned with defining who is responsible for inconsistency handling, specifying when consistency checks should be performed and generally configuring the inconsistency management process for a particular project. Since these activities are concerned with concrete models on a project, we consider them to be instance-level activities. The remaining phases shown in Figure 5.16 also concern the instance level.

Detection of inconsistencies is followed by their diagnosis. For instance, inconsistencies can be diagnosed according to their source, cause and impact. Measurement can additionally be applied to the entire set of inconsistencies to provide an overview for the modeler and to facilitate benchmarking. The diagnostic information is supposed to assist the modeler during the subsequent handling of inconsistencies, which includes inconsistency resolution. The term “handling” is used instead of “resolution”, because during this phase the modeler may decide to tolerate some inconsistencies by ignoring them or deferring their resolution. The choices made during the inconsistency handling are tracked and profiled to provide the modeler with additional information on later runs of the inconsistency management phases.

The embedding of our solution into the inconsistency management phases is shown in Figure 5.17. The steps performed by the tool developer, the modeler and the fully automatic steps are distinguished in the diagram.

It can be seen that our solution spans all the phases shown in Figure 5.16, except for the detection of overlaps, which was addressed in Chapter 4. Additionally, we distinguish a phase called “development of resolution types”, which is necessary if automated support for inconsistency resolution is to be offered to the modeler.

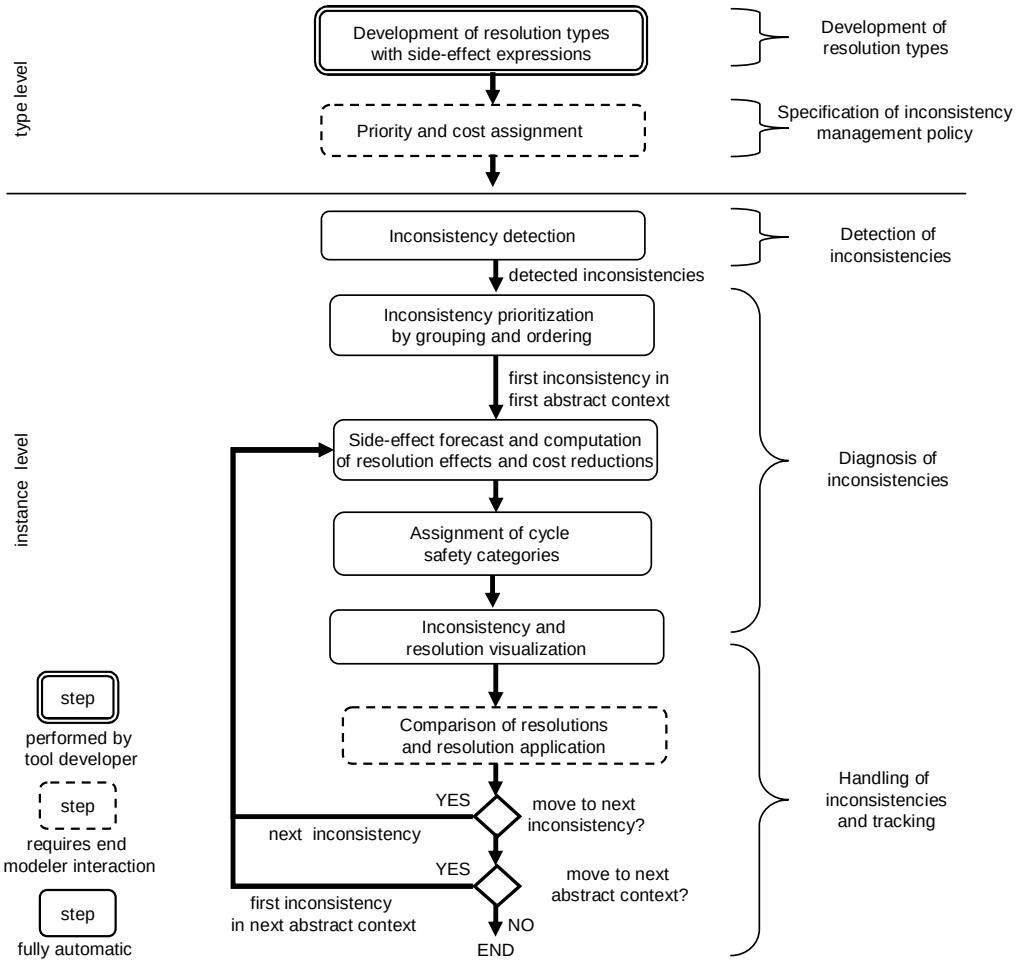


Figure 5.17: Augmented inconsistency management process

5.8 Summary and Discussion

In this chapter, we presented our solution to address several challenges confronting the modeler during inconsistency resolution. First of all, we described how a set of detected inconsistencies can be automatically prioritized to minimize the number of context-switches that the modeler has to perform while resolving these inconsistencies. Furthermore, we presented an approach to forecasting resolution side-effects to assist the modeler in analyzing the impact of resolutions for a given inconsistency. We also showed that the forecasted side-effects are instrumental in quantifying the relative advantages of alternative resolutions for one inconsistency. We demonstrated that the comparison of alternative resolutions can be further enhanced with the assignment of different costs to inconsistency types. Finally, we presented an approach to determining cycle safety of resolutions, also based on side-effect forecast.

Our presented solution was illustrated using the domain of process and object life cycle models, although it also generalizes to inconsistency resolution for other types of models. The solution can potentially be realized in different modeling tools and embedded into a specific inconsistency management process, as was shown in Section 5.7.

In the beginning of this chapter, we explained that existing approaches to the detection of resolution side-effects are performed on the type level and therefore only provide

approximations of the side-effects for a given resolution at the instance level. In our solution, concrete side-effects are forecasted by the evaluation of side-effect expressions. Of course, the accuracy of the side-effect forecasts depends on the correctness and completeness of the side-effect expressions. We imagine that automated support would be very valuable to assist the tool developer in attaining a correct and complete set of side-effect expressions. For example, the tool developer could use the existing type-level analysis techniques [Mens et al., 2006a, Mens et al., 2006b] to ensure that all the dependencies between inconsistency and resolution types are covered by the side-effect expressions.

Similarly, our approach to resolution cycle detection can benefit from integration with existing type-level analyses such as [Mens et al., 2006a, Mens et al., 2006b], which could be used to automatically compute the resolution type trees. Even with such an enhancement, our detection approach is not complete and in the worst case all the resolutions are placed into the safety-unknown category. By utilizing the concept of an inconsistency history [Wagner et al., 2003], we can ensure that all resolution cycles are detected at least a number of steps in advance and not once the cycle has already occurred. How much in advance the cycles are detected depends on the lookahead value that is used.

Using the results presented in Chapter 4 and this chapter, inconsistencies can be identified and resolved in a given set of process and object life cycle models. This already forms an essential part of our framework for integrating process and object life cycle modeling. Certain scenarios also require transformations between process and object life cycle models, which is the topic of the following chapter.

Chapter 6

Model Transformations

We begin this chapter by explaining the scenarios that require model transformations between process and object life cycle models in Section 6.1. Motivated by these scenarios, we identify several requirements that these transformations should satisfy. We then present our proposed model transformations from process to object life cycle models, which we call object life cycle extraction, in Section 6.2 and from object life cycle to process models, which we refer to as process model generation, in Section 6.3. For each transformation, we show how the identified requirements are addressed.

6.1 Model Transformation Requirements

In Model-Driven Engineering (MDE), a *model transformation* is defined as "the automatic generation of a target model from a source model, according to a transformation definition" [Kleppe et al., 2003]. A transformation definition usually comprises a description of how elements of the source modeling language are used to create elements of the target modeling language. Although MDE approaches especially emphasize model transformations from a higher level of abstraction to a lower one, transformations within the same abstraction level are also common and especially useful in supporting multi-view modeling scenarios [Sendall and Kozaczynski, 2003, Giese and Wagner, 2008]. In the context of process and object life cycle models, we are concerned with transformations within the same abstraction level, since we integrate these models as complementary views on the same application.

Since object life cycle models are being developed to serve as best practices or even standards in certain industries (e.g. ACORD[1], IAA[2], HL7[3]), consistency of process models that represent business processes of some organization in an industry with such object life cycle models is an emerging requirement. One approach to achieving consistency of a process model with a given object life cycle model is to first put the models through a consistency check and then resolve the identified inconsistencies. This is illustrated in Figure 6.1(a), where consistency of the process model PM and the object life cycle model OLC is the goal. However, this approach can lead to a lengthy inconsistency resolution process challenged by context-switches, resolution side-effects and resolution cycles (cf. Chapter 5). As an alternative, a transformation can be applied to automatically

[1] http://www.acord.org
[2] http://www.ibm.com/industries/financialservices/doc/content/solution/278918103.html
[3] http://www.hl7.org

construct a process model from the given object life cycle model such that consistency is ensured by construction, as illustrated in Figure 6.1(b).

Apart from achieving consistency of process models with independently developed object life cycle models, it may be similarly required that all process models within the same organization manipulate objects of a particular type in a consistent manner. In such a scenario, the process model that should serve as reference for all other process models can be used to extract a so-called reference object life cycle model, which can then be used to align the remaining process models. This inverse transformation, illustrated in Figure 6.1(c), should ensure that the produced object life cycle model is consistent with the original process model.

From these two model transformation scenarios, we infer our first requirement that *the transformations between process and object life cycle models should ensure the consistency of the source and target models (R1)*.

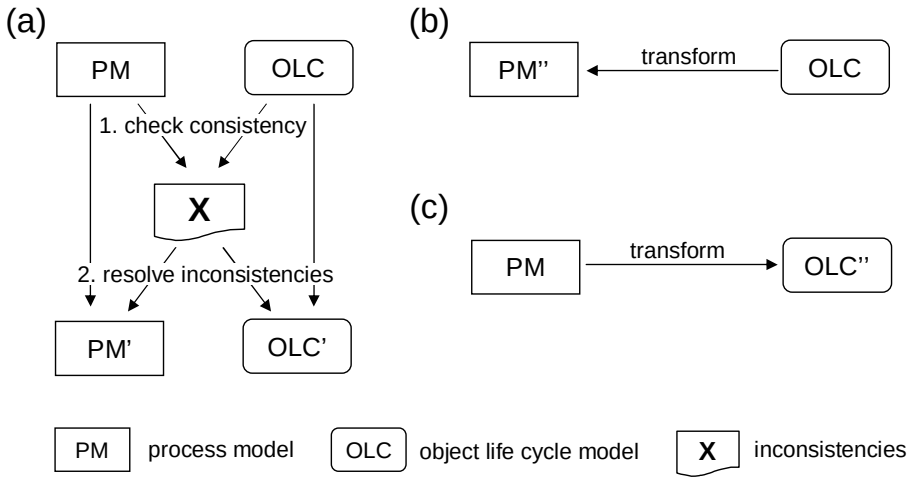


Figure 6.1: Obtaining consistency

The process and object life cycle model consistency defined in Chapter 4 gives rise to a many-to-many relation between process and object life cycle models, i.e. many object life cycle models are potentially consistent with one given process model and vice versa. As a result, PM' in Figure 6.1(a) is not necessarily the same as PM'' in Figure 6.1(b) even if only the process model was adapted during the inconsistency resolution shown in (a). Both PM' and PM'' may be consistent with OLC, even if PM' contains more control nodes or more actions than PM''.

Instead of defining transformations that produce an arbitrary target model, as long as it is consistent with the source model, it is desirable to produce a target model that is unique with regards to a certain property. In our case, it is desirable to produce models that are minimal with respect to a particular measure of size, such that no irrelevant model elements that encumber the target model are introduced.

The particular measure of size could be different for each modeling language. Assuming a set $\mathcal{C}_M$ of all models consistent with a given model M, the chosen measure of size should be used to define a partial order on the elements of $\mathcal{C}_M$. Then, *the transformations between process and object life cycle models should produce the minimal element of $\mathcal{C}_M$ with regards to the defined partial order given M as the source model (R2)*.

Apart from transforming process models to object life cycle models with the final goal of obtaining consistency, this transformation is also useful for generating a more abstract view on the original process model, which can then be used for inferring some proper-

ties about the behavior of the process model. Consider for instance temporal properties that concern the states of one object type only, such as "objects of this type always reach a particular state". It may be easier to check whether such properties hold by examining the object life cycle model for that object type, as opposed to a large process model that deals with many different object types. In such scenarios, it is required that *the object state sequences produced by the execution of the process model are exactly the same as the object state sequences of the object life cycle model produced by the transformation (R3)*.

In the following sections, we present transformations from process models to object life cycle models and vice versa, which we respectively call *object life cycle extraction* and *process model generation*. Both transformations are performed as a sequence of processing steps, which we primarily describe using pseudocode. Some of the steps of these transformations could also be specified in one of the existing model transformation languages, such as those provided in VIATRA [Csertán et al., 2002], GReAT [Agrawal, 2004] or QVT [QVT, 2008]. Apart from showing the details of the transformation definitions, we discuss how each transformation addresses the identified requirements.

6.2 Object Life Cycle Extraction

The extraction of object life cycle models from process models is performed in several steps, as illustrated in Figure 6.2. At each step, an intermediate model that is semantically closer to an object life cycle model is produced, until finally the object life cycle model itself is created. Process models with repository or routed data flow can be taken as the input to this transformation.

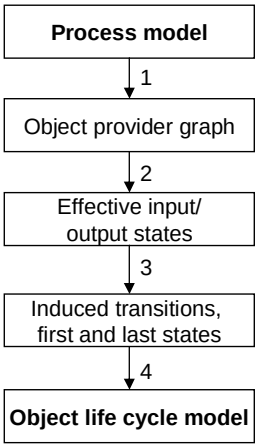


Figure 6.2: Object life cycle extraction steps

The computation of the object provider graph and the effective input and output states (steps 1 and 2) were described in Chapter 4, where they were initially used to evaluate the correctness of a state specification in a process model. The derivation of induced transitions, first and last states from the effective input and output states (step 3) was also described in Chapter 4 for defining consistency conditions for process and object life cycle models. In step 4 of the transformation, the object life cycle model is constructed based on the information about the induced transitions, first and last states. This is described in the algorithm in Listing 6.1.

Listing 6.1: constructOLC()

```
1 ObjectLifeCycleModel constructOLC(ProcessModel pm, ObjectType t)
2   ObjectLifeCycleModel olc = new ObjectLifeCycleModel();
3   for each( Transition tran in pm.getInducedTransitions(t)) do
4     olc.addTransition(tran.getSourceState(), tran.getTargetState());
5   for each(State first in pm.getFirstStates(t)) do
6     olc. addInitialTransitionTo ( first );
7   for each(State last in pm.getLastStates(t)) do
8     olc.addFinalTransitionFrom( last );
9   return olc;
```

An example of applying the object life cycle extraction is shown in Figure 6.3. The claims handling process model shown in (a) is used to compute the object provider graph shown in (b). For each node in the object provider graph, the computed effective input and output states are also shown. The process model shown in (c) illustrates the computed induced transitions, first and last states. Finally, the object life cycle model for the $Claim$ object type shown in (d) is constructed. This result is obtained when the addTransition(State s1, State s2) function used in line 4 in Listing 6.1 only adds a transition from state s1 to state s2 to the constructed object life cycle model if such a transition does not already exist. Since consistency of the produced object life cycle model with the original process model considers their object state sequences only and not the events labeling transitions, these events can be arbitrarily assigned during the transformation. The construction of object life cycle models can be repeated for all object types manipulated by the original process model.

6.2.1 Ensuring Consistency

It is straightforward to show that object life cycle models produced by the object life cycle extraction are consistent with the original process model by demonstrating that all consistency conditions in Definition 55 of Chapter 4 are satisfied. For example, transition conformance holds, because there is a transition created in the produced object life cycle model for every induced transition in the process model (lines 3 and 4 in Listing 6.1). Since no transition can be created in the object life cycle model without a corresponding induced transition in the process model, transition coverage also holds. In a similar manner, we can show that the first and last state conformance, and the initial and final transition coverage also hold. This demonstrates that the proposed object life cycle extraction approach satisfies requirement R1.

6.2.2 Target Model Minimality

To address requirement R2, we evaluate the minimality of an object life cycle model extracted from a given process model by comparing it to other object life cycle models that are consistent with the process model. To this end, we define the *object life cycle inclusion* relation on object life cycle models as follows.

Definition 73 (Object life cycle inclusion)**.** *Given two object life cycle models* $OLC_t = (S, s_\alpha, s_\omega, \Sigma, \delta)$ *and* $OLC'_t = (S', s'_\alpha, s'_\omega, \Sigma', \delta')$ *for object type t, we say that* OLC'_t includes OLC_t*, written* $OLC_t \leq_{inc} OLC'_t$*, if and only if for each two states* $s_1, s_2 \in S'$*,* $|\{e \in \Sigma \mid s_2 \in \delta(s_1, e)\}| \leq |\{e \in \Sigma' \mid s_2 \in \delta'(s_1, e)\}|$.

According to the above definition, an object life cycle model OLC'_t includes another object life cycle model OLC_t if for each ordered pair of states in OLC'_t, there are fewer

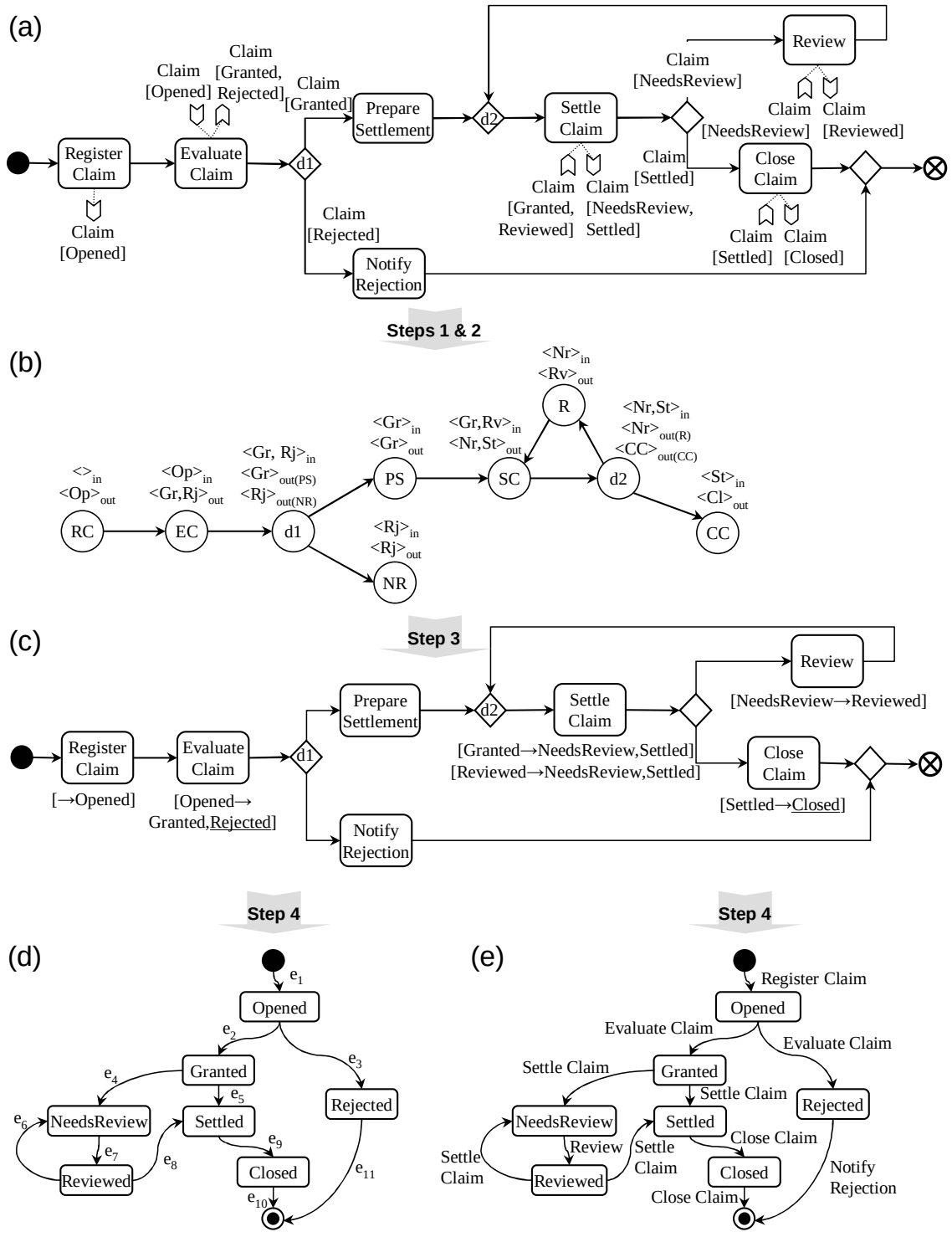


Figure 6.3: Object life cycle extraction example

or the same number of transitions in OLC_t than there are in OLC'_t. The object life cycle inclusion relation is a partial order, since it is reflexive, antisymmetric and transitive.

In the following, we show the minimality of object life cycle models produced by the object life cycle extraction, therefore confirming that R2 is satisfied for this transformation. Since the steps of the object life cycle extraction are described using pseudocode and natural language, we provide a proof sketch as opposed to a formal proof for the following theorem.

Theorem 5. *Let a workflow graph $G = (N, E)$ with either repository or routed data flow using a set of object types T and a state specification be given, and let $OLC_t = (S, s_\alpha, s_\omega, \Sigma, \delta)$ be the object life cycle model produced by the object life cycle extraction from G for object type $t \in T$. With respect to object life cycle inclusion, OLC_t is the minimal element of the set of all object life cycle models for object type t that are consistent with G, i.e. there is no other OLC'_t consistent with G where $OLC'_t \leq_{inc} OLC_t$.*

Proof sketch: We use proof by contradiction to show this. Suppose that there exists an $OLC'_t = (S', s'_\alpha, s'_\omega, \Sigma', \delta')$ consistent with G where $OLC'_t \leq_{inc} OLC_t$ and $OLC'_t \neq OLC_t$. Then, there must exist $s_1, s_2 \in S$ where $|\{e \in \Sigma' \mid s_2 \in \delta'(s_1, e)\}| \leq |\{e \in \Sigma \mid s_2 \in \delta(s_1, e)\}|$. Since OLC_t only has one transition between every pair of states by construction, this means that there is no transition from s_1 to s_2 in OLC'_t. However, since there is a transition from s_1 to s_2 in OLC_t, it means that one of the following must be true: (1) (a, s_1, s_2) is an induced transition for t in G for some action a, (2) $s_1 = s_\alpha$ and s_2 is a first state of t in G, or (3) $s_2 = s_\omega$ and s_1 is a last state of t in G. In turn, this means that either (1) transition conformance, (2) first state conformance, or (3) last state conformance does not hold for OLC'_t and G. In this case, OLC'_t and G are not consistent, and therefore we reach a contradiction. □

We have now shown that the object life cycle extraction satisfies requirements R1 and R2. Since the produced object life cycle models can be used as a complementary view on the application, it is also important to consider whether this view contains enough information that may be valuable to the user of this view. Apart from extracting the state evolution protocol satisfied by the process model, the produced object life cycle model can also be seen as an abstract view on the process model only showing how the milestones for one particular object type are achieved in that process. In this case, it is also valuable for the user to see which actions are responsible for inducing state transitions on objects of the object type in question. To this end, actions associated with induced transitions, first and last states can be used to generate events labeling transitions in the produced object life cycle model, as shown in Figure 6.3(c). Furthermore, knowing if there are multiple ways of achieving a particular milestone may also be of importance. To distinguish these in the produced object life cycle model, each induced transition in the process model should give rise to a uniquely labeled transition in the produced object life cycle model. However, extending the object life cycle extraction to support this would no longer guarantee the minimality of the produced object life cycle models with respect to the object life cycle inclusion. A more relaxed form of object life cycle inclusion can be defined to express the minimality of the produced models.

6.2.3 Behavior Preservation

As already discussed in the beginning of this chapter, certain scenarios require that the produced object life cycle models represent behavior that is equivalent to that captured in

the original process model. One such scenario is using the extracted object life cycle models to check temporal properties. Suppose for example that we want to check whether the $Claim$ objects always reach the state $Closed$ in the claims handling process model in Figure 6.3(a). By examining the extracted object life cycle model for $Claim$, it is clear that this property does not hold, because if a $Claim$ reaches the state $Rejected$, it transits to the final state without reaching the $Closed$ state. Therefore, we conclude that the property does not hold for the original process model. However, to be able to argue in this manner, it is required that the object state sequences produced by the execution of the process model are exactly the same as the object state sequences of the object life cycle model. This property is analogous to *trace equivalence* of models known in the literature on concurrent system modeling [van Glabbeek, 1990]. The property indeed holds for the example shown in Figure 6.3, however generally it is not guaranteed by the object life cycle extraction.

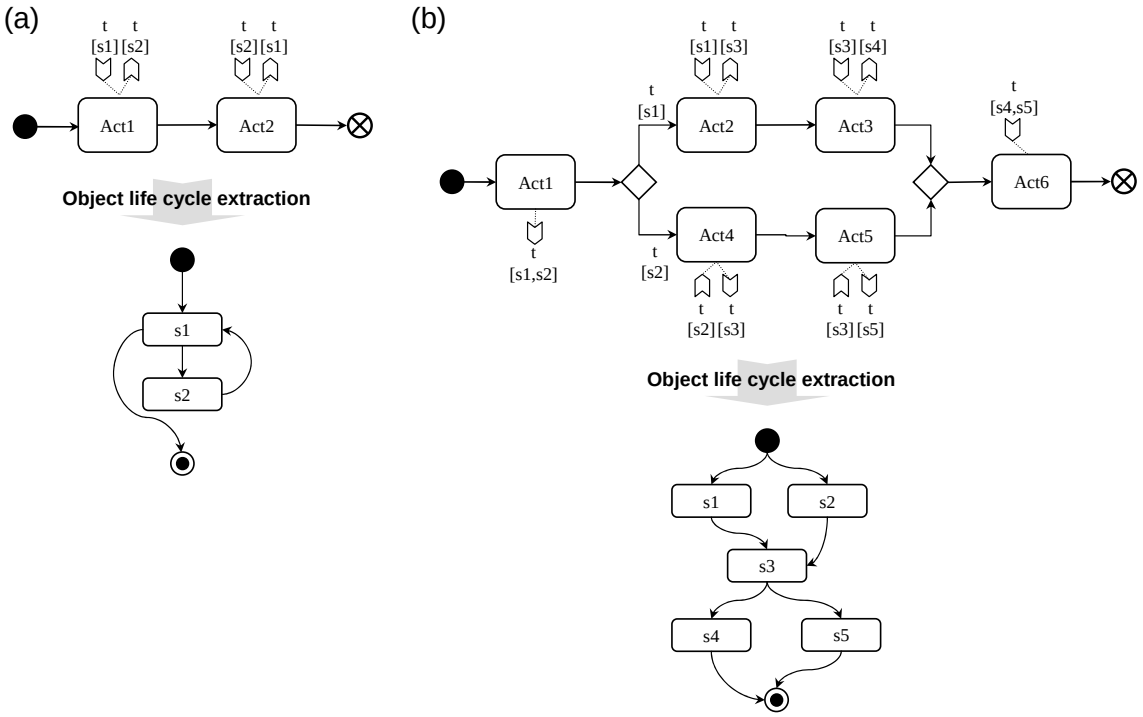


Figure 6.4: Additional state sequences introduced during extraction

The object state sequences produced by the executions of the process model are always contained in the object state sequences of the object life cycle models produced by the extraction. However, the opposite does not hold, because the produced object life cycle models may have additional object state sequences. This is illustrated with two examples in Figure 6.4. The process model in (a) produces one object state sequence for object type t, namely $\langle s1, s2, s1\rangle$. However, the object state sequences of the produced object life cycle model include $\langle s1, s2, s1, s2\rangle$, $\langle s1, s2, s1, s2, s1, s2\rangle$, etc. The process model in (b) gives rise to the object state sequences $\langle s1, s3, s4\rangle$ and $\langle s2, s3, s5\rangle$ for object type t. Apart from these, the object state sequences of the produced object life cycle model also contain $\langle s1, s3, s5\rangle$ and $\langle s2, s3, s4\rangle$.

Additional object state sequences are introduced during the object life cycle extraction whenever there exist two induced transitions (a_1, s_1, s_2) and (a_2, s_2, s_3) in the given process model where a_1 is not an object provider of a_2 with respect to t. In such a case,

we say that action a_2 is a *fictitious successor* of action a_2 and call s_2 a *deceptive state*. For instance, the process model in Figure 6.4(a) gives rise to induced transitions $(Act2,\ s2,\ s1)$ and $(Act1,\ s1,\ s2)$ for object type t, but $Act2$ is not an object provider of $Act1$ with respect to t. Therefore, $Act1$ is a fictitious successor of $Act2$ and $s2$ is a deceptive state. The process model shown in Figure 6.4(b) has two actions with fictitious successors: $Act5$ is a fictitious successor of $Act2$ and $Act3$ is a fictitious successor of $Act4$.

To ensure that the process model and the object life cycle models extracted from it are equivalent with respect to object state sequences, we propose a pre-processing of the state specification in the process model prior to the object life cycle extraction. In the pre-processing, deceptive states are relabeled to eliminate all fictitious successors. The algorithm for the pre-processing is given in Listing 6.2.

Listing 6.2: preProcess()

```
1  preProcess(ProcessModel pm, ObjectType t)
2    for each (InducedTransition tran1 in pm.computeInducedTransitions(t)) do
3      for each (InducedTransition tran2 in pm.computeInducedTransitions(t)) do
4        if (tran1.getTargetState() == tran2.getSourceState() &
5            !tran1.getAction(). isObjectProvider(tran2.getAction(), t))
6          Action act = tran1.getAction();
7          State deceptiveState = tran1.getTargetState();
8          State relabeledState = relabel(deceptiveState);
9          act.getProducedStates(t). replace(deceptiveState, relabeledState);
10         for each (Node n in pm.getNodes) do
11           if (act. isObjectProvider(n, t))
12             if (n. isAction())
13               n.getAcceptedStates(t). add(relabeledState);
14             else if (n. isDecision())
15               for each (Edge e in n.getOutgoingEdges()) do
16                 if (e.getCondition(t). contains(deceptiveState))
17                   e.getCondition(t).add(relabeledState);
```

During the pre-processing, whenever an action act is determined to have a fictitious successor with respect to a deceptive state deceptiveState, this state is first of all relabeled in the produced states of act for t. The relabeling is performed to produce a new state that does not yet appear in the state specification of the given process model. To ensure the correctness of the state specification, accepted and produced states of nodes for which act is an object provider with respect to t are also adjusted. For actions that have act as an object provider with respect to t, the new state relabeledState is added to their accepted state sets for t. For decisions that have act as an object provider with respect to t, relabeledState is added to those conditions associated with their outgoing edges that contain deceptiveState. This ensures the satisfaction of the syntactic correctness conditions defined in Definition 47 in Chapter 4.

Figure 6.5 shows the results of the pre-processing on the process models from Figure 6.4 and how they affect the produced object life cycle models. In (a), the deceptive state $s1$ is relabeled to $s1'$ in the produced states of action $Act2$. This removes the cycle from the produced object life cycle model, thereby removing the additional object state sequences. In (b), the deceptive state $s3$ is relabeled to $s3'$ in the produced states of $Act4$ and $s3'$ is added to the accepted states of $Act5$ to maintain a correct state specification. As a result of this relabeling, targeted to eliminate $Act3$ as the fictitious successor of $Act4$, $Act5$ also ceases to be a fictitious successor of $Act2$, because $(Act5,\ s3,\ s5)$ is no longer an induced transition of t after the relabeling.

We have now presented our approach to transforming process models to object life cycle models and showed how it addresses the requirements identified in the beginning

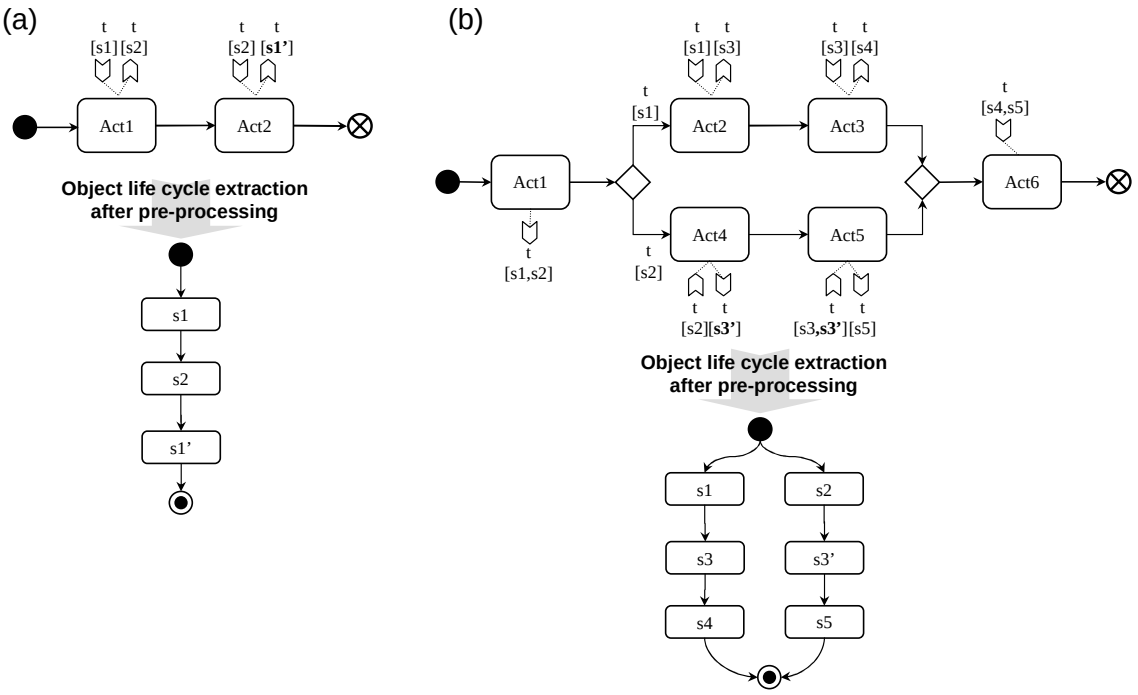


Figure 6.5: Object life cycle extraction after pre-processing

of the chapter. In the following section, we describe our proposed approach to the inverse transformation.

6.3 Process Model Generation

As explained in the beginning of this chapter, the main scenario for process model generation is obtaining consistency with a given object life cycle model that represents a best practice, a standard or an organizational reference. Since one process model usually deals with several object types and may therefore require consistency with more than one object life cycle model, process model generation needs to handle multiple object life cycle models as input. The approach to process model generation presented in this section has also been described in one of our earlier publications [Küster et al., 2007].

Up to this point in the dissertation, we have considered consistency of a process model with one object life cycle model that represents a state evolution protocol for objects of a particular type. Since state evolution of objects is not always independent from one another, given several object life cycle models, it may not always be appropriate to simply enforce object life cycle conformance and coverage of a given process model with respect to all of these object life cycle models. Consider for example the object life cycle models for $Claim$ and $Payment$ object types in Figure 6.6(a) and (b). The payment should only be created if the claim is granted, and a claim can be settled only once the full payment has been made. A process model that satisfies object life cycle conformance and coverage with respect to these two object life cycle models individually can still lead to undesirable execution states, for example where the $Claim$ is in state $Settled$ and the $Payment$ is in state $PartiallyPaid$. Object dependencies need to be taken into account to avoid such situations. This can be done by specifying additional constraints on the given object life cycle models that express synchronization of object state evolution to form a joint state

evolution protocol.

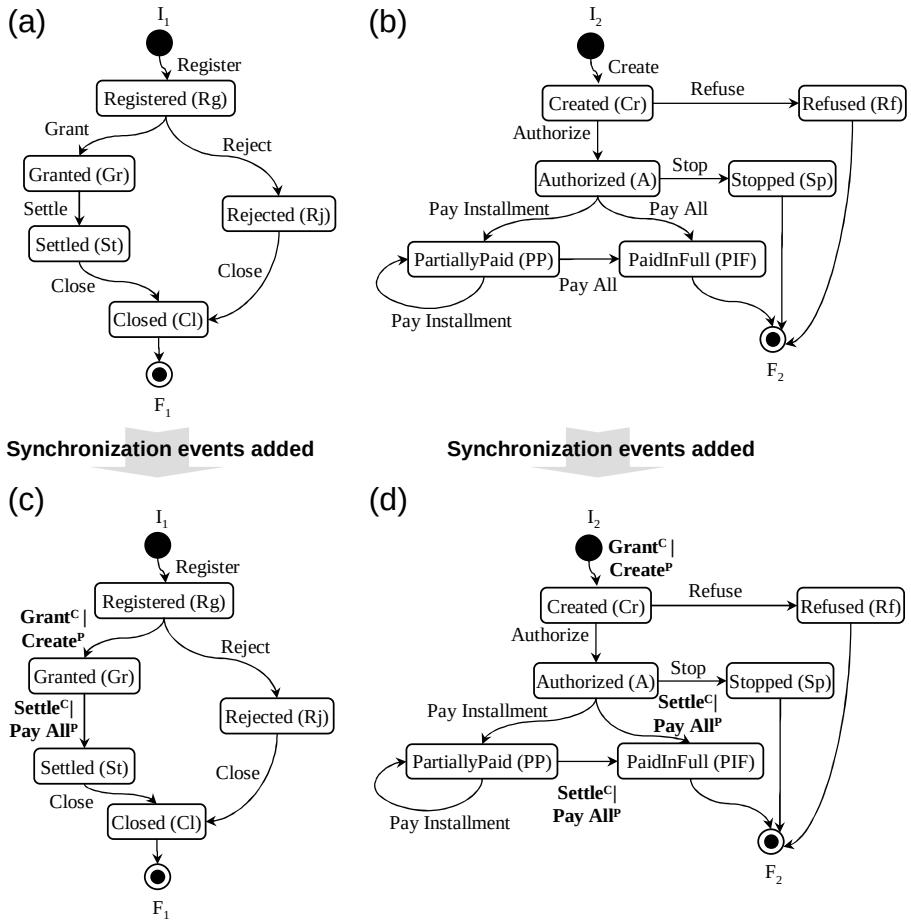


Figure 6.6: Object life cycle models for (a) Claim and (b) Payment, (c) and (d) after adding synchronization events

Our approach to process model generation begins with the identification and specification of the required synchronization of the given object life cycle models. This step is performed manually, followed by several automatically performed steps shown in Figure 6.7. In step 2, a composition of the object life cycle models that represents the joint state evolution protocol is computed based on the original object life cycle models and the constraints defined for their synchronization. In step 3, the composite object life cycle model is used to generate a set of actions for the process model and the order in which these actions should appear in the process model. Each distinct event labeling a transition in the composite object life cycle model is used to generate an action for the final process model. In step 4, actions are combined into process fragments, where they are additionally connected to decisions and merges. Finally, the process fragments are connected to produce the resultant process model in step 5. In the following, we focus on the generation of process models with repository data flow, later explaining how routed data flow can be introduced subsequently to the generation. The details of each process model generation step are explained next.

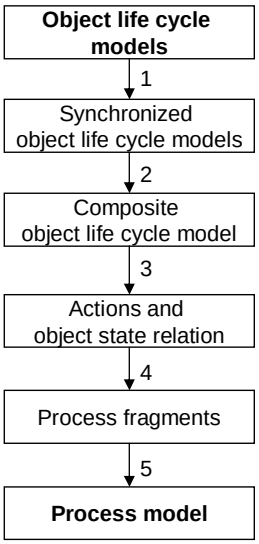


Figure 6.7: Process model generation steps

6.3.1 Synchronization and Composition of Object Life Cycle Models

A number of different mechanisms for capturing synchronization of several behavioral models are known from the literature on automata theory and concurrent system modeling. These include synchronized transitions or actions [Arnold, 1994, Hopcroft et al., 2006], explicit modeling of communication [Brand and Zafiropulo, 1983] and timed synchronizations [Alur and Dill, 1992]. Other custom means of expressing synchronization constraints, for example using the Object Constraint Language (OCL) [Kleppe and Warmer, 2003], could also be used. Our interpretation of object life cycle models does not intuitively extend to enabling direct communication between models, since each object life cycle model represents a protocol rather than an executing and communicating component. The approach of synchronized transitions, established in automata theory, does however naturally extend into the context of object life cycle models. Therefore, this is the approach that we take to specifying synchronization of object life cycle models.

In the following, we define synchronization and subsequent composition of *two* object life cycle models. In the case when more than two object life cycle models are given, further iterations of the synchronization and composition need to be applied, each time using the composite object life cycle model produced by the previous iteration and a new object life cycle model. To identify synchronized transitions in two object life cycle models, those transitions that should be triggered at the same time and result in objects of both types transiting to new states need to be distinguished. Such transitions are labeled with the same event, called a *synchronization event*, to indicate that they are required to synchronize.

Definition 74 (Synchronization event)**.** *Given two object life cycle models* $OLC_{t_1} = (S_1, s_{\alpha_1}, s_{\omega_1}, \Sigma_1, \delta_1)$ *and* $OLC_{t_2} = (S_2, s_{\alpha_2}, s_{\omega_2}, \Sigma_2, \delta_2)$*, an event* $e \in \Sigma_1 \cap \Sigma_2$ *is called a* synchronization event*.*

Suppose that we wish to generate a new claims handling process model from the object life cycle models for $Claim$ and $Payment$ shown in Figure 6.6(a) and (b). Since the payment should only be created if the claim is granted, we create a synchronization event

called $Grant^C \mid Create^P$ and use it to replace the $Grant$ and $Create$ events in the $Claim$ and $Payment$ object life cycle models, respectively, as shown in Figure 6.6(c) and (d). Furthermore, a claim can be settled only once the full payment has been made and thus we introduce another synchronization event called $Settle^C \mid Pay\ All^P$ to replace the $Settle$ and $Pay\ All$ events in the $Claim$ and $Payment$ object life cycle models, also shown in Figure 6.6(c) and (d). Note that to avoid spurious synchronization events, we should ensure that the intersection of the event sets of the given object life cycle models is empty prior to introducing synchronization events in to the models.

Once synchronization events are introduced, the composition of object life cycle models is computed. We use the following definition for the composition of two object life cycle models, based on the definitions of the *product automaton* [Hopcroft et al., 2006] in automata theory:

Definition 75 (Composition, composite object life cycle model)**.** *A* composition *of two object life cycle models* $OLC_{t_1} = (S_1, s_{\alpha_1}, s_{\omega_1}, \Sigma_1, \delta_1)$ *and* $OLC_{t_2} = (S_2, s_{\alpha_2}, s_{\omega_2}, \Sigma_2, \delta_2)$ *is a* composite object life cycle model $OLC = (S_1 \times S_2, (s_{\alpha_1}, s_{\alpha_2}), (s_{\omega_1}, s_{\omega_2}), \Sigma_1 \cup \Sigma_2, \delta)$*, where:*

$$\delta((s_1, s_2), e) = \begin{cases} \delta_1(s_1, e) \times \delta_2(s_2, e) & \text{if } e \in \Sigma_1 \cap \Sigma_2 \\ \delta_1(s_1, e) \times \{s_2\} & \text{if } e \in \Sigma_1 \setminus \Sigma_2 \\ \{s_1\} \times \delta_2(s_2, e) & \text{if } e \in \Sigma_2 \setminus \Sigma_1 \end{cases}$$

Given a composite object life cycle model $OLC = (S, s_\alpha, s_\omega, \Sigma, \delta)$ that is a composition of object life cycle models for object types $t_1, ..., t_n$ and a state $s = (s_1, ..., s_n) \in S$, we can refer to the state of an individual object type t contained in s as $s[t]$, i.e. $s[t_i] = s_i$ for $1 \leq i \leq n$. One object life cycle model is considered to be a trivial composition of just itself.

Figure 6.8 shows the composition of the $Claim$ and $Payment$ object life cycle models after they were augmented with synchronization events. In the diagram, states and events are marked with superscripts C and P to reflect that they belong to $Claim$ and $Payment$, respectively.

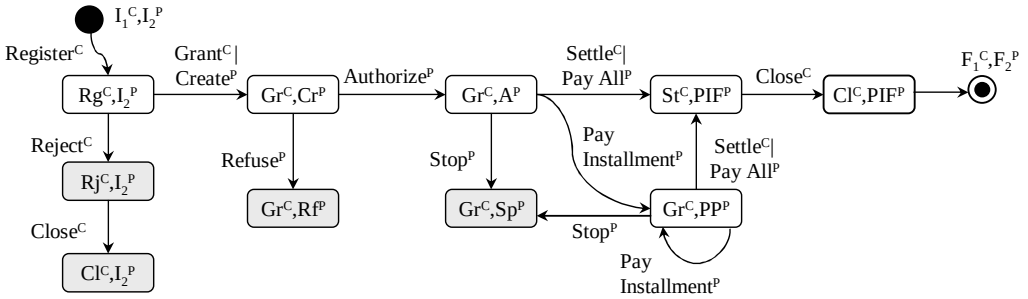


Figure 6.8: Composition of the Claim and Payment object life cycle models

Figure 6.8 shows only those states that are reachable from the composite initial state I_1^C, I_2^P. Two composite states leading to the composite final state, Cl^C, Rf^P and Cl^C, Sp^P, are not reachable in the composite object life cycle model. Additionally, the states highlighted in gray do not lead to the composite final state. These are essentially deadlock states and it is arguable whether they should be used for the process model generation. For example, the transition trace $Register^C$, $Reject^C$, $Close^C$ seems valid and should be reflected in the generated process model, even though the final state of the $Payment$ object is not reached. The trace $Register^C$, $Grant^C \mid Create^P$, $Refuse^P$ also seems valid,

although it may not be desirable that the $Claim$ is still in state Gr ($Granted$) even though the $Payment$ has been refused. Such phenomena are typical if the composed object life cycle models were created independently from each other, which may often be the case in practice.

In some scenarios, the original object life cycle models can be adjusted to ensure that a composite final state is always reachable. For example, a new transition from state $Granted$ to $Rejected$ can be added to the $Claim$ object life cycle model, labeled with a new synchronization event $Refuse^C \mid Refuse^P$, also used to replace the $Refuse$ event in the $Payment$ object life cycle model. Such an adjustment would ensure that the composite final state Cl^C, Rf^P is reachable from Gr^C, Rf^P via the state Rj^C, Rf^P. Therefore, in a realization of this transformation, it will be valuable to inform the modeler of the traces that do not lead to a composite final state and let him/her either adjust the object life cycle models or choose to exclude these traces from the generated process model. In the following, we use all the states reachable from the composite initial state for the generation of the process model. To ensure that a composite object life cycle model is a valid object life cycle model (cf. Definition 39), we add an additional transition from every state that has no outgoing transitions to the final state.

Up till now, we have discussed synchronization and composition with the goal of generating a process model that deals with one object of each given type, e.g. the claims handling process model we wish to generate should deal with one $Claim$ and one $Payment$. In some cases, the process model may need to manipulate multiple objects of the same type. If the number of objects of object type t is known to be n in advance, then the corresponding object life cycle model OLC_t should be first copied n times to produce models $OLC_{t_1}, ..., OLC_{t_n}$. These resulting object life cycle models should be used as any other models during the synchronization and composition steps. For example, if there should always be two $Payments$ for one $Claim$, two object life cycle models for $Payment$ should be used for process model generation. In essence, each $Payment$ is then treated as a distinct object type, which also makes it possible to define different synchronization events for different $Payments$.

In the following section, we describe the remaining steps 3, 4 and 5 of the process model generation.

6.3.2 Process Model Construction

In step 3, given a composite object life cycle model $OLC = (S, s_\alpha, s_\omega, \Sigma, \delta)$ that is a composition of object life cycle models for object types $t_1, ..., t_n$, we iterate over all its transitions and create a set of actions N_A. Each created action $a \in N_A$ matches one of the following object manipulation operations with respect to each object type t_i where $1 \leq i \leq n$ (cf. Section 3.2.3 of Chapter 3):

- *create:* a has only a data output of type t_i;
- *update:* a has a data input and a data output of type t_i;
- *no impact:* a has no data inputs or outputs of type t_i.

The action generation algorithm is given in Listing 6.3. For each transition, the object manipulation operation is determined with respect to each object type (lines 4-14). The set of already created actions is then checked for an action that matches the event labeling the current transition and the overall pattern comprising the determined object

manipulation operations (line 15-20). If such an action exists, then only its state specification is augmented (lines 28,32-33). Otherwise, a new action associated with the event of the current transition is created (line 19) and its data inputs, data outputs and a state specification are added according to the pattern of object manipulation operations (lines 25-33).

Listing 6.3: generateActions()

```
1 Set generateActions(CompositeOLC olc)
2   Set actions = new Set();
3   for each ( Transition tran in olc.getTransitions()) do
4     // determine object manipulation operation for current transition
5     Pattern objManipulationOp = new Pattern();
6     for each (ObjectType t in olc.getTypes()) do
7       State s1 = tran.getSource().getState(t);
8       State s2 = tran.getTarget().getState(t);
9       if (s1 != t. getInitial () & s1 != s2)
10        objManipulationOp.add(t,"update");
11      else if (s1 == t. getInitial () & s1 != s2)
12        objManipulationOp.add(t,"create");
13      else if (s1 == s2)
14        objManipulationOp.add(t,"no impact");
15    // if there is no matching action, create one
16    boolean noMatch = false;
17    Action act = getMatchingAction(actions, tran.getEvent(), objManipulationOp);
18    if (act == null)
19      act = actions.addNew(tran.getEvent());
20      noMatch = true;
21    // add data inputs, data outputs and state specification to action
22    for each (ObjectType t in olc.getTypes())
23      State s1 = tran.getSource().getState(t);
24      State s2 = tran.getTarget().getState(t);
25      if (objManipulationOp.get(t) == "create" | objManipulationOp.get(t) == "update")
26        if (noMatch)
27          act.addDataOutput(t);
28        act.addToProducedStates(t,s2);
29      if (objManipulationOp.get(t) == "update")
30        if (noMatch)
31          act.addDataInput(t);
32        act.addToAcceptedStates(t,s1);
33        act.addToDependencyStateSet(t,s1,s2);
34  return actions;
```

Figure 6.9 illustrates how two transitions from the composite object life cycle model for the $Claim$ and $Payment$ object types shown in Figure 6.8 are processed by the algorithm in Listing 6.3. In (a), it is first determined that the transition from Gr^C, A^P to St^C, PIF^P corresponds to the *update* object manipulation operation for both $Claim$ and $Payment$. Assuming that the action set N_A is empty at this point in the algorithm, a new action labeled $Settle^C \mid Pay\ All^P$ is created. Data inputs and outputs of both object types are created for this action. The accepted and produced states, and the dependency state sets are assigned to reflect the state changes corresponding to the transition in the composite object life cycle model. In (b), another transition labeled $Settle^C \mid Pay\ All^P$ is processed, which also matches the *update* object manipulation operation for both object types. Since an action with this label and object manipulation operations already exists, the state specification for this action is augmented with new information. In this case, PP is added to the accepted states of this action for the $Payment$ object type and a new dependency state set is introduced (marked in bold).

Once the set of actions N_A is generated, the order in which they should appear in the process model needs to be determined. To achieve this, we define the following relation on the action set that identifies direct predecessors and successors of actions based

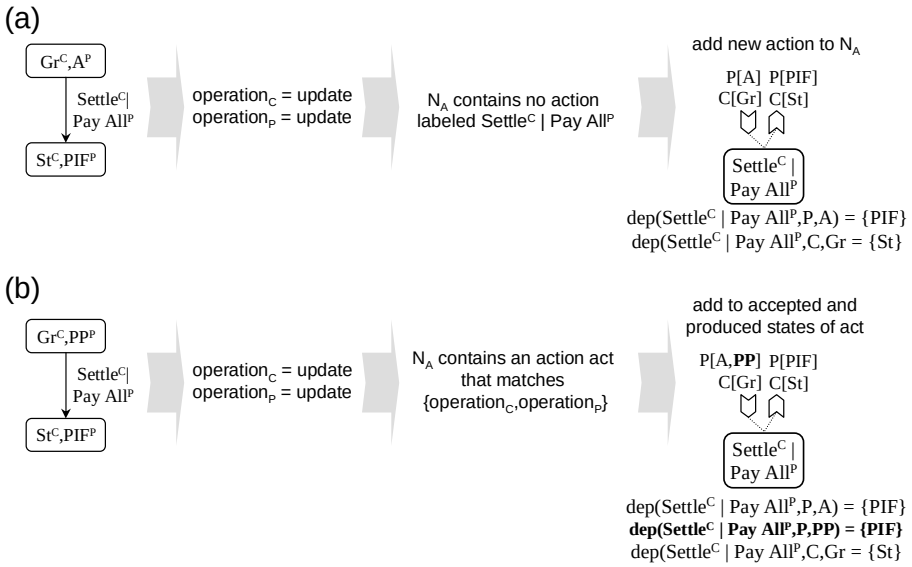


Figure 6.9: Example action generation

on their data inputs and outputs, as well as their accepted and produced state sets for different object types.

Definition 76 (Object state relation)**.** *Given an action $a_1 \in N_A$ with data outputs of types $t_{11}, ..., t_{1k}$ and an action $a_2 \in N_A$ with data inputs of types $t_{21}, ..., t_{2m}$, a_1 is a predecessor of a_2 in the* object state relation*, written $a_1 \prec a_2$, if and only if for all object types $t \in \{t_{11}, ..., t_{1k}\} \cap \{t_{21}, ..., t_{2m}\}$, $prod(a_1, t) \bigcap acpt(a_2, t) \neq \emptyset$.*

For the set of actions created from the composition of the object life cycle models for $Claim$ and $Payment$ shown in Figure 6.8, the object state relation among other elements includes the following: $Register^C \prec Reject^C$, $Reject^C \prec Close^C$, $Settle^C \mid Pay\ All^P \prec Close^C$, $Authorize^P \prec Settle^C \mid Pay\ All^P$ and $Pay\ Installment^P \prec Settle^C \mid Pay\ All^P$. Note that the object state relation is not a partial order, as it is not transitive. In other words, $Register^C \not\prec Close^C$ even though $Register^C \prec Reject^C$ and $Reject^C \prec Close^C$. The computation of the object state relation can be carried out in a straightforward manner by checking the intersections of the accepted and produced state sets of different actions in N_A.

In step 4, the action set and the computed object state relation are used to generate *process fragments*. Three types of process fragments are generated: action fragments, start fragments and stop fragments (illustrated in Figure 6.10), as described in the algorithm in Listing 6.4. First, the algorithm iterates over the actions in N_A, represented by the set actions in the pseudocode (line 4). For each action act, the numbers of its predecessors and successors in the object state relation are examined and based on these an appropriate action fragment is generated. If act has more than one predecessor, it is preceded by a merge in the action fragment, so that in the final process model multiple edges from the predecessor nodes can be merged into one edge connected to act. If act has more than one successor, it is followed by a decision in the action fragment. A decision is also added to the action fragment when act has only one successor, but it produces last states, which means that for at least one object type t some of the states in its produced state set for t have a transition to a final state of t (see Appendix A for detailed pseudocode). In this case, the decision will split the edge from act into two edges, one leading to its successor node and the other to the stop node.

After the iteration over actions, the start and stop fragments are generated (lines 14-24). The start fragment contains the start node, which is succeeded by a decision if there is more than one action without a predecessor in the object state relation. On the other hand, the stop fragment contains the stop node, which is preceded by a merge if there is more than one action that produces last states.

Listing 6.4: generateFragments()

```
1 Set generateFragments(Set actions, ObjectStateRelation rel)
2   Set fragments = new Set();
3   // generate action fragments
4   for each (Action act in actions) do
5     ProcessFragment frag = new ProcessFragment(act);
6     if (rel.getPredecessors(act).size() > 1)
7       Merge merge = new Merge();
8       merge.connectTo(act);
9     if (rel.getSuccessors(act).size() > 1 |
10        (rel.getSuccessors(act).size() == 1 & act.producesLastStates()) )
11      act.connectTo(new Decision());
12    fragments.add(frag);
13  // generate start fragment
14  StartNode start = new StartNode();
15  ProcessFragment startFrag = new ProcessFragment(start);
16  if (rel.getActionsWithNoPredecessors().size() > 1)
17    start.connectTo(new Decision());
18  fragments.add(startFrag);
19  // generate stop fragment
20  StopNode stop = new StopNode();
21  ProcessFragment stopFrag = new ProcessFragment(stop);
22  if (getActionsProducingLastStates(actions).size() > 1)
23    Merge merge = new Merge();
24    merge.connectTo(stop);
25  fragments.add(stopFrag);
26  return fragments;
```

In the final step 5, process fragments are connected according to the algorithm given in Listing 6.5. Once again, we iterate over the actions in N_A and use the object state relation to determine how the generated fragments should be connected.

Listing 6.5: connectFragments() and Fragment.connectTo()

```
1 connectFragments(Set actions, ObjectStateRelation rel, Set fragments)
2   for each (Action act in actions) do
3     Set preds = rel.getPredecessors(act);
4     if (preds.size() == 0)
5       getStartFragment(fragments).connectTo(getFragment(fragments,act));
6     else
7       for each (Action pred in preds) do
8         getFragment(fragments,pred).connectTo(getFragment(fragments,act));
9   Set lasts = getActionsProducingLastStates(actions);
10  for each (Action last in lasts) do
11    getFragment(fragments,last).connectTo(getStopFragment(fragments));
12
13 connectTo(Fragment frag)
14  Edge new = this.getLastNode().connectTo(frag.getFirstNode());
15  if (this.getLastNode() instanceof Decision & !frag.isStopFragment())
16    new.assignEdgeCondition(this.getAction().getProdStates(). intersect (frag.getAction().getAcptStates()));
17  else if (this.getLastNode() instanceof Decision & frag.isStopFragment())
18    new.assignEdgeCondition(this.getAction(). getLastStates());
```

As described in the fragment connection algorithm in Listing 6.5, connection of two process fragments is performed as follows. If the predecessor fragment contains no decision, a new edge is simply created to connect the last node of the predecessor fragment to

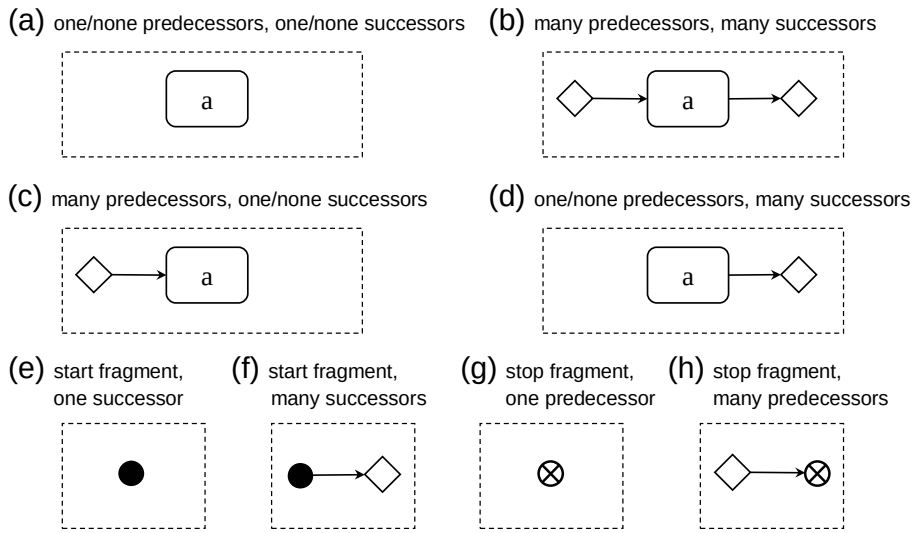


Figure 6.10: Fragment types: (a)-(d) action fragments (e)-(f) start fragments (g)-(h) stop fragments

the first node of the successor fragment. The connection is established in the same way if there is a decision in the predecessor fragment, but an additional edge condition is added to the new edge that connects the decision to the first node of the successor fragment.

Figure 6.11 shows an example of three action fragments generated for our claims handling process model. The action $Reject^C$ has one predecessor ($Register^C$) and one successor ($Close^C$), and hence its containing action fragment has no decisions or merges. On the other hand, $Close^C$ and $Settle^C \mid Pay\ All^P$ both have several predecessors and one successor, which results in action fragments that contain a merge, but no decision. The connection of these fragments is also illustrated in the diagram.

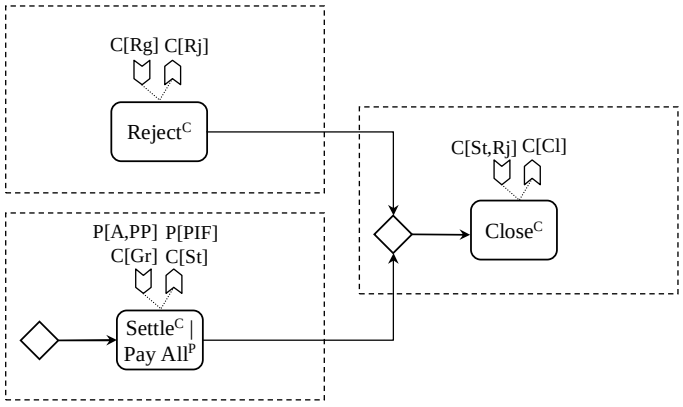


Figure 6.11: Example fragment connection

The complete process model generated from the $Claim$ and $Payment$ object life cycle models (see Figure 6.6(a) and (b)) is shown in Figure 6.12(a). The transformation can of course also be applied just for one object type, for example Figure 6.12(b) shows the process model generated from the $Claim$ object life cycle model only.

Repository data flow in a generated process model can be transformed to routed data flow by leaving the control flow as is and adding typed edges to connect each action with its object providers with respect to each object type. Decision and merge nodes need to

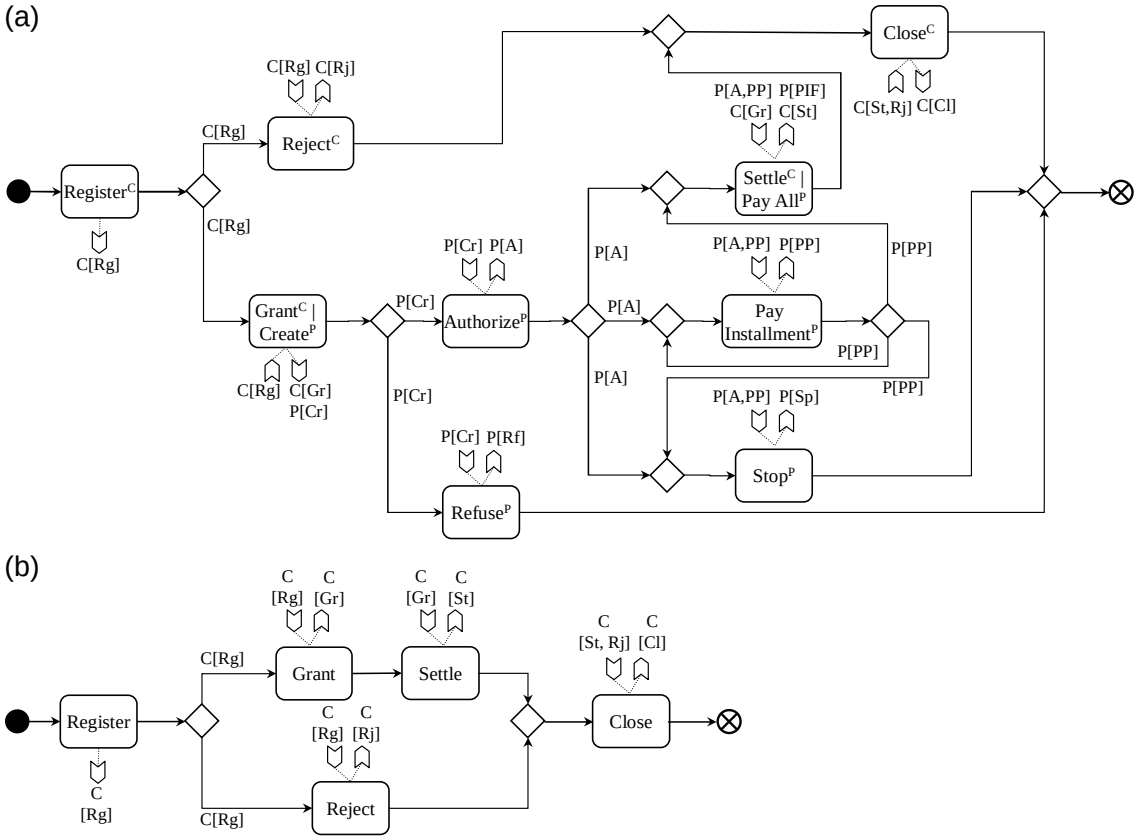


Figure 6.12: Generated process models: (a) from both Claim and Payment object life cycle models (b) from Claim object life cycle model

be introduced to cover the cases where one action has more than one object provider or is an object provider to more than one other action with respect to a particular object type. Some control-flow edges may be turned into typed edges to avoid cluttering the model with redundant edges and control nodes. This is illustrated using a subset of the generated claims handling process model in Figure 6.13.

In the presence of forks and joins, transforming repository data flow to routed data flow would be much more intricate, since for each case where data flow needs to be split (joined), we would need to determine whether to use a decision or a fork (a merge or a join). An inappropriate choice of node could result in a model that is not sound, by either introducing a deadlock or a lack of synchronization (cf. Section 4.2 of Chapter 4). Since the process model generation presented here does not introduce forks or joins, the transformation is simplified.

6.3.3 Ensuring Consistency

In our approach to process model generation, dependencies between a set of given object life cycle models are explicitly taken into account during the first two steps of the transformation, which result in a composite object life cycle being computed. Therefore, the generated process model is required to be consistent with the composite object life cycle model, and not necessarily with each of the original object life cycle models. To show that consistency is ensured, we first reformulate the syntactic consistency conditions given in Definition 55 of Chapter 4 such that they apply to a composite object life cycle model.

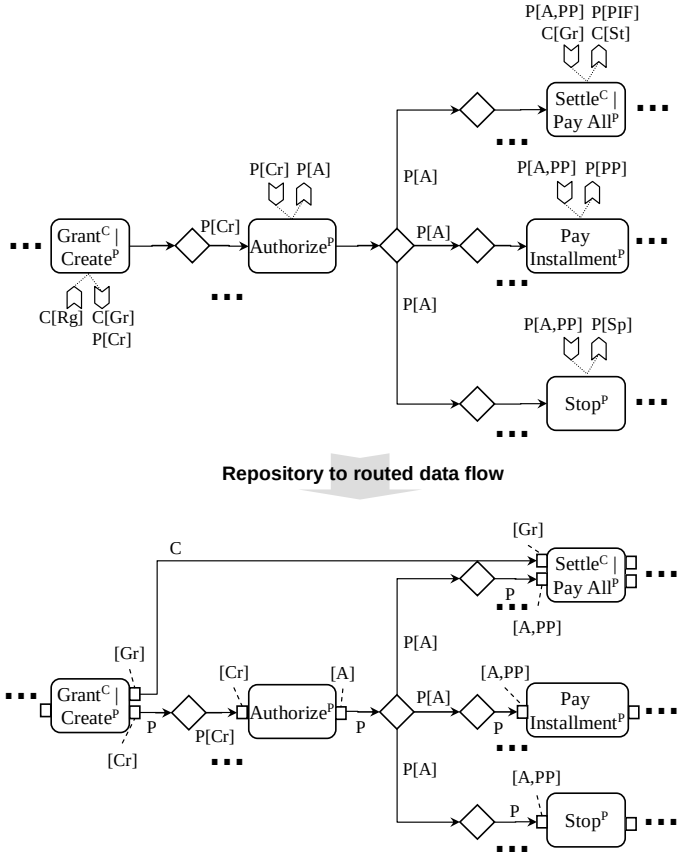


Figure 6.13: Example of transforming repository data flow to routed data flow

Definition 77 (Syntactic consistency conditions: extended for composite object life cycle models)**.** *Let a workflow graph $G = (N, E)$ with either repository or routed data flow using a set of object types T and a state specification be given. Furthermore, let a composite object life cycle model $OLC = (S, s_\alpha, s_\omega, \Sigma, \delta)$ that is a composition of object life cycle models for object types $t_1, ..., t_n$ be given. We define the following consistency conditions for G and OLC:*

C55.1: for each induced transition (a, s_1, s_2) of t_i in G where $1 \leq i \leq n$, there exists a transition from s'_1 to s'_2 in OLC such that $s'_1[t_i] = s_1$ and $s'_2[t_i] = s_2$ (transition conformance);

C55.2: for each first state s of t_i in G where $1 \leq i \leq n$, there exists a transition from s_α to s' in OLC such that $s'[t_i] = s$ (first state conformance);

C55.3: for each last state s of t_i in G where $1 \leq i \leq n$, there exists a transition from s' to the final state s_ω in OLC such that $s'[t_i] = s$ (last state conformance);

C55.4: for each transition from a state s_1 to a state s_2 in OLC and for each t_i where $1 \leq i \leq n$, if $s_1[t_i] \neq s_2[t_i]$, $s_1[t_i] \neq s_\alpha[t_i]$ and $s_2[t_i] \neq s_\omega[t_i]$ then there exists an induced transition $(a, s_1[t_i], s_2[t_i])$ of t_i in G for some action $a \in N$ (transition coverage);

C55.5: for each transition from a state s_1 to a state s_2 in OLC and for each t_i where $1 \leq i \leq n$, if $s_1[t_i] = s_\alpha[t_i]$ then $s_1[t_i]$ is a first state of t_i in G (initial transition coverage);

C55.6: for each transition from a state s_1 to a state s_2 in OLC and for each t_i where $1 \leq i \leq n$, if $s_2[t_i] = s_\omega[t_i]$ then $s_2[t_i]$ is a last state of t_i in G (final transition coverage).

The following theorem states that the composite object life cycle model computed in step 2 of the generation and the produced process model are consistent. Since the steps of the process model generation are primarily described using pseudocode, we provide a proof sketch as opposed to a formal proof for this theorem.

Theorem 6. *Let a composite object life cycle model $OLC = (S, s_\alpha, s_\omega, \Sigma, \delta)$ that is a composition of object life cycle models for object types $t_1, ..., t_n$ be given and let $G = (N, E)$ be the workflow graph generated from OLC. All consistency conditions defined in Definition 77 are satisfied for G and OLC and therefore they are consistent.*

Proof sketch. In the following, proof sketches are given to show that each consistency condition holds. Let t be any of the object types $t_1, ..., t_n$.

C55.1 (transition conformance): Given any induced transition (a, s_1, s_2) of t in G, we know that $s_1 \in acpt(a, t)$, $s_2 \in prod(a, t)$ and $s_2 \in dep(a, t, s_1)$ for some action $a \in N$. Since a has both a data input and a data output of type t, it matches the *update* object manipulation operation for t and must have been generated from some transition $s'_1 \xrightarrow{e} s'_2$ in OLC where $s'_1[t] = s_1$ and $s'_2[t] = s_2$.

C55.2 (first state conformance): Given any first state s of t in G, we know by definition that $s \in prod(a, t)$ for some action $a \in N$ that does not have a data input of type t. Action a matches the *create* object manipulation operation for t and must have been generated from some transition $s_\alpha \xrightarrow{e} s'$ in OLC where $s'[t] = s$.

C55.3 (last state conformance): Given any last state s of t in G, we know that $s \in prod(n, t)$ for some action or decision $n \in N$, from which there is a path p to the stop node such that p does not contain any actions or decisions with data outputs of type t. This also means that $s \in prod(a, t)$ for some action $a \in N$ (it is possible that $a = n$), from which there is a path p' to the stop node such that p' does not contain any actions with data outputs of type t. The process fragment containing a must have been connected to a stop process fragment by lines 9-11 in Listing 6.5 and producesLastStates() used in Listing 6.4 (see Appendix A for details) must be true for a. This means that there is a transition $s' \xrightarrow{e} s_\omega$ in OLC where $s'[t] = s$.

C55.4 (transition coverage): Given any transition $s_1 \xrightarrow{e} s_2$ in OLC where $s_1[t] \neq s_2[t]$, $s_1[t] \neq s_\alpha[t]$ and $s_2[t] \neq s_\omega[t]$, we know that $s_1[t] \in acpt(a, t)$, $s_2[t] \in prod(a, t)$ and $s_2[t] \in dep(a, t, s_1[t])$ for some action $a \in N$. Since s_1 is not an initial state of OLC, it must be a target of some transition used to generate an action a' where $s_1[t] \in prod(a', t)$. There must be a node n such that $n \lhd_t a$ and n is either a' itself or it is a decision on a path from a' to a. By construction, $effout(n, a, t) = prod(n, t)$ and hence $s_1[t] \in effin(a, t)$. Since $s_2[t] \in effout(a, n', t)$ for some node $n' \in N$ and $s_1[t] \neq s_2[t]$, $(a, s_1[t], s_2[t])$ is an induced transition of t in G.

C55.5 (initial transition coverage): Given any transition $s_1 \xrightarrow{e} s_2$ in OLC where $s_1[t] = s_\alpha[t]$, we know that $s_2[t] \in prod(a, t)$ for some action $a \in N$ that has no data inputs of type t. Hence, $s_2[t]$ is a first state of t in G.

C55.6 (final transition coverage): Given any transition $s_1 \xrightarrow{e} s_2$ in OLC such that $s_2[t] = s_\omega[t]$, we know that $s_1[t] \in prod(a, t)$ for some action $a \in N$ such that producesLastStates() used in Listing 6.4 (see Appendix A for details) is true for a. By the connection of process fragments, there must be a path p from a to the stop node. Let n be either the action a itself or the last decision that occurs on path p before the stop node. By the assignment of edge conditions during the connection of process fragments, $s_1[t] \in prod(n, t)$. Since by construction $effout(n, n', t) = prod(n, t)$ for some node $n' \in N$, $s_1[t]$ is a last state of t in G. □

6.3.4 Target Model Minimality

To address requirement R2, we consider the minimality of a generated process model in terms of the number of actions that it contains.

We note that it may be possible to reduce the total number of actions in a generated process model by merging some actions and assigning dependency state sets to ensure that the merged actions still lead to the required induced transitions. An example of such a merger is shown in Figure 6.14, where two actions from the generated process model shown in Figure 6.12(b) are merged into one action that would be executed twice. Such merger would lead to multiple events in the given object life cycle models[4] being integrated to produce one action. However, we assume that the events in the object life cycle models indicate the desired decomposition into actions and therefore consider such merger to be undesirable.

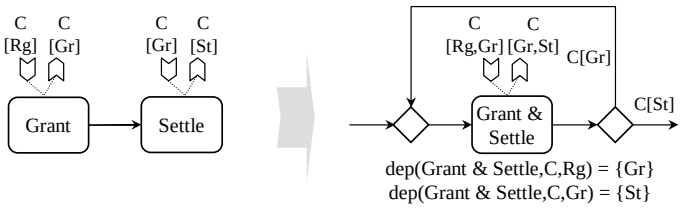


Figure 6.14: Example of merging actions

By construction, each action in a process model produced by the process model generation is associated with exactly one event (cf. Listing 6.3). Nevertheless, it is possible that a generated process model contains more than one action per event. For target model minimality, we ensure that the process model generation does not introduce unnecessary actions for each event in the given object life cycle models.

We define the *process inclusion* relation to compare a generated process model with other process models that are consistent with the composite object life cycle model obtained from the object life cycle models used as the input for the transformation.

Definition 78 (Process inclusion)**.** *Given a set of actions N_A generated from a composite object life cycle model $OLC = (S, s_\alpha, s_\omega, \Sigma, \delta)$, we define a function $event : N_A \to \Sigma$ to map an action $a \in N_A$ to an event $e \in \Sigma$ that labels the transition from which a was generated. Given a composite object life cycle model $OLC = (S, s_\alpha, s_\omega, \Sigma, \delta)$, and two workflow graphs $G = (N, E)$ and $G' = (N', E')$, we say that G'* includes *G, written $G \leq_{inc} G'$, if and only if for each event $e \in \Sigma$, $|\{a \in N \mid event(a) = e\}| \leq |\{a' \in N' \mid event(a') = e\}|$.*

According to the above definition, a process model G' includes another process model G if for each distinct event in the given composite object life cycle model, there are less or the same number of actions associated with that event in G than there are in G'. The process inclusion relation is a partial order, since it is reflexive, antisymmetric and transitive.

In the following, we confirm that R2 is satisfied for the process model generation by showing the minimality of the produced process models. Minimality is guaranteed provided that the original object life cycle models do not contain multiple transitions with the same source and target states. Similarly as for other theorems in this chapter, we provide a proof sketch for the following theorem.

[4]Here, we consider the events in the object life cycle models after synchronization events have been introduced in step 1 of the process model generation.

Theorem 7. *Let a composite object life cycle model $OLC = (S, s_\alpha, s_\omega, \Sigma, \delta)$ that is a composition of object life cycle models for object types $t_1, ..., t_n$ and the workflow graph $G = (N, E)$ generated from OLC be given. Let us further assume that for each object type t_i where $1 \leq i \leq n$ and two transitions $s_1 \xrightarrow{e_1} s_2$ and $s_3 \xrightarrow{e_2} s_4$ in OLC, $(s_1[t_i] = s_3[t_i] \wedge s_2[t_i] = s_4[t_i]) \Rightarrow e_1 = e_2$. With respect to process inclusion, G is the minimal element of the set of all process models that have at least one action associated with each event in OLC and are consistent with OLC, i.e. there is no other G' consistent with OLC where $G' \leq_{inc} G$.*

Proof sketch: We use proof by contradiction to show this. Suppose that there exists a workflow graph $G' = (N', E')$ that has at least one action associated with each event in OLC and is consistent with OLC. Furthermore, $G' \leq_{inc} G$ and $G' \neq G$. This means that there exist two actions $a_1, a_2 \in N$ associated with some event $e \in \Sigma$ that are integrated into one action $a' \in N'$. Actions a_1 and a_2 must match different patterns of object manipulation operations, cf. lines 15-20 in Listing 6.3, which means a_1 and a_2 must match different operations for at least one object type t_i where $1 \leq i \leq n$. There are three cases to consider, since each of the generated actions can match one of three object manipulation operations, namely *create*, *update* and *no impact* (cf. Listing 6.3).

Case 1: Actions a_1 and a_2 match *create* and *update* operations for object type t_i (regardless of which action matches which operation). Merging these actions into one action a' requires a' to have either only a data output of type t_i or both a data input and a data output of type t_i. In the former case, induced transitions associated with a_2 would no longer be induced, which would violate transition coverage. In the latter case, first states of t_i associated with a_1 would no longer be first states, which would violate initial transition coverage. Therefore, this case leads to a contradiction.

Case 2: Actions a_1 and a_2 match *create* and *no impact* operations for object type t_i. Assuming that $event(a_1) = event(a_2) = e$, there must be at least two transitions in OLC labeled with e. If e is not a synchronization event, e must label more than one transition in the object life cycle model OLC_{t_i} and not label any transitions in any other of the given object life cycle models. Then, every action that is associated with e should match a *create* or an *update* operation for t_i. Otherwise, if e is a synchronization event, then transitions of OLC_{t_i} labeled e must synchronize with transitions in some other object life cycle model. Every transition labeled e in the composite object life cycle model OLC changes the state of t_i. Therefore, every action in the generated process model that is associated with e should match a *create* or an *update* operation for t_i. This is a contradiction, since a_2 matches the *no impact* operation for t_i.

Case 3: Actions a_1 and a_2 match *update* and *no impact* operations for object type t_i. This case also leads to a contradiction according to the same argumentation as in Case 2. □

A generated process model can be subsequently customized to the specific needs of an organization. Possible customization steps are parallelization of actions, addition of extra data inputs to actions that read objects without changing their state and addition of supplementary actions. Subprocesses can also be factored out of the generated process model, such that each subprocess covers different parts of the original object life cycle models. Our example demonstrates that the generation can produce process models with non-deterministic decision nodes (see Figure 6.12), in which case refinement of decision logic needs to be done as part of the customization. Checking consistency with the original object life cycle models after the customization phase is necessary, provided that the changes introduced during customization of the process model are not restricted. Alternatively, only consistency-preserving changes could be allowed during customization.

6.4 Summary and Discussion

In this chapter, we presented two transformations for process and object life cycle models, namely the object life cycle extraction and the process model generation. As demonstrated, both transformations comprise several non-trivial transformation steps. In addition to presenting the details of the transformations themselves, we also showed that they produce target models with several desirable properties. Target models produced by the transformations are not only consistent with the source models, but are also minimal with regards to the set of all models consistent with a given source model. The minimality was shown with respect to the introduced object life cycle inclusion and process inclusion relations. For object life cycle extraction, we additionally showed how behavior preservation can be ensured by pre-processing the given process model.

The issue related to fictitious successors and deceptive states in the context of object life cycle extraction is related to the so-called *implied scenarios* that occur during the synthesis of state machines from scenarios [Uchitel et al., 2001, Muccini, 2002]. The goal of scenario synthesis is to generate state machines for different objects that participate in a set of given scenario specifications [Whittle and Schumann, 2000, Uchitel et al., 2003]. Under certain conditions, it is possible that a synthesized state machine contains behaviors that are not valid with respect to the original scenario specifications, which are called implied scenarios. As opposed to seeing this as an undesirable phenomenon, Uchitel et al discuss how implied scenarios can be used to elaborate behavioral specifications [Uchitel et al., 2004]. A similar approach may be interesting in the context of object life cycle extraction.

Our process model generation technique does not introduce concurrency into the produced process models. Given a set of object life cycle models that do not synchronize at all or only have few synchrozations, it is possible that the produced process model contains a large number of edges, decisions and merges that capture many possible action interleavings. We suggest parallelization of actions as a possible manual customization of a generated process model. The process model generation could also potentially be enhanced to identify concurrency automatically using techniques from the *theory of regions* and *Petri net synthesis* [Ehrenfeucht and Rozenberg, 1989, Badouel and Darondeau, 1998], which addresses the problem of obtaining a Petri net from a transition system.

The described transformations facilitate the scenarios described in the beginning of this chapter, such as attaining consistency of process models with reference object life cycle models and using object life cycle models as views that abstract from some of the details captured in a process model. Together with the consistency checking and inconsistency resolution, described in Chapters 4 and 5, these transformations complete our solution for the integration of process and object life cycle modeling during the analysis and design phases of Business Process Management (BPM). In the following chapter, we explore the transition from the design phase to the implementation phase.

Chapter 7

From Design to Implementation

In Chapters 3-6, we have presented our solution for the integration of process and object life cycle modeling based on consistency checking, inconsistency resolution and model transformations. In this solution, process and object life cycle models are considered as complementary views on an application used during the analysis and design phases in the Business Process Management (BPM) life cycle. In this chapter, we investigate the transition from the design phase to the implementation phase, with a special focus on object-centric approaches to process implementation.

Object-centric approaches distribute process control-flow logic among several synchronizing components, where each component represents the life cycle of a particular object. In this chapter, we describe how such components relate to the object life cycle models we have considered so far and demonstrate how a complete decomposition of control flow captured in a process model among several components can be achieved. Furthermore, we identify the management of component coupling as one of the main challenges in object-centric process implementations, since high coupling makes it difficult to distribute, maintain and adapt the components. To tackle this challenge, we propose a technique for statically analyzing a given process model to compute the expected component coupling. This coupling forecast facilitates direct adaptation of the process model to alleviate the component coupling before actually deriving the implementation components.

We begin this chapter by providing an overview of object-centric process implementation approaches in Section 7.1. In Section 7.2, we introduce our selected language for object-centric process implementation, namely the Business State Machines (BSMs) [Beers and Carey, 2006], and present our chosen metric for measuring component coupling. In Section 7.3, we demonstrate how process models can be mapped to BSMs and identify the process model constructs that contribute to the component coupling. These observations are formalized in Section 7.4, where we describe how the expected coupling is computed based on a given process model. Finally, we discuss the generalization of our approach in Section 7.5. Note that the contents of this chapter are based on one of our earlier publications [Wahler and Küster, 2008].

7.1 Object-Centric Process Implementation

Most existing languages for process modeling (e.g. EPCs, UML Activity Diagrams and BPMN) and process implementation (e.g. BPEL) can be considered *task-centric* or *activity-centric*, because they represent processes as a set of tasks connected by control-flow el-

ements to indicate the order of task execution. As has already been pointed out in Chapter 2 (see Section 2.5.2), a line of alternative object-centric approaches to process implementation has been proposed in the recent years (e.g. [van der Aalst et al., 2001, Nandi and Kumaran, 2005]).

Object-centric approaches distribute process control-flow logic among several *components*. Each component in an object-centric process implementation is described by what essentially is an object life cycle model. However, object life cycle models are given an execution semantics in this context, as opposed to being treated as state evolution protocols. This distinction is similar to that made in UML State Machines between behavioral state machines and protocol state machines (cf. Section 2.3.2 of Chapter 2). In the following, we use the term "object life cycle component" or simply "component" to refer to executable object life cycle models in order to avoid confusion with our usual understanding of an object life cycle model in terms of the syntax and semantics defined in Section 3.3 of Chapter 3.

Synchronization of components ensures that the overall process logic is correctly implemented, as illustrated in Figure 7.1(a). Components can be synchronized based on the well-known synchronization mechanisms from automata theory and concurrent system modeling, which include synchronized transitions or actions [Arnold, 1994, Hopcroft et al., 2006], explicit modeling of communication [Brand and Zafiropulo, 1983] and timed synchronizations [Alur and Dill, 1992]. When considering object life cycle models as state evolution protocols from which a process model is to be generated, we employed the synchronized transitions mechanism for expressing synchronization (cf. Section 6.3 of Chapter 6). In object-centric process implementation, however, synchronization by explicit communication between components is the approach that is most commonly used, since this facilitates distribution of components. For example, communication is used to synchronize components in the following two object-centric process implementation approaches: proclets [van der Aalst et al., 2001] and Adaptive Business Objects (ABOs) [Nandi and Kumaran, 2005]. In the following, we take a closer look at proclets, ABOs and other existing object-centric approaches to process implementation.

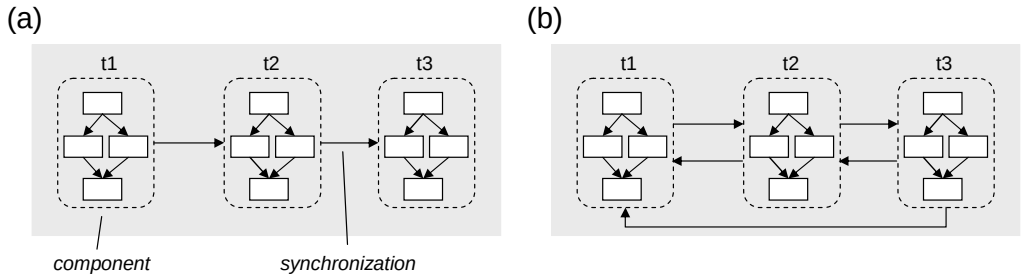


Figure 7.1: Synchronization of components in object-centric process implementations

7.1.1 Existing Object-Centric Approaches

Proclets have been presented by van der Aalst et al [van der Aalst et al., 2001] as a framework for modeling lightweight interacting workflows. The authors describe proclets as "objects equipped with an explicit life-cycle" [van der Aalst et al., 2001] and use workflow nets to represent them. Explicit communication channels are used to enable the interaction between proclets and hence their synchronization. Communication channels are described by a set of properties that facilitate different types of communication, such

as multicast/broadcast, push/pull and synchronous/asynchronous. At run time, proclets register with a naming service that facilitates the correlation of communicated messages.

Messages exchanged between proclets are called performatives and are associated with a number of attributes, including time, information about the message sender and receivers, and the message content. Several different types of performatives are distinguished to indicate the purpose of the message, for example the purpose could be an acknowledgement or a request.

ABOs have been proposed by Nandi et al [Nandi and Kumaran, 2005] as the implementation and execution model for the so-called *artifact-centric* approach to the development of BPM applications [Nigam and Caswell, 2003, Bhattacharya et al., 2005, Liu et al., 2007]. An ABO encapsulates behavioral and structural information about a business object (also referred to as a business artifact). The behavior is represented with a finite state machine, where every transition is labeled according to the event-condition-action paradigm. The structure of the data relevant for the ABO is captured in a data graph, while only references to data locations and not the data themselves are stored in the ABO.

Each ABO has an external interface, through which it receives events from the outside. Once an event that has an associated transition in the current ABO state is received, the condition of that transition is evaluated and if it evaluates to true then the action is executed and the current state is updated. A transition action can either be a data action that performs create/read/update/delete operations on the ABO data, or it can be a remote action that communicates with other ABOs or any other remote application. Synchronization of ABOs is thus implemented as communication between ABOs inside the remote actions associated with state transitions.

Data-driven process structures [Müller et al., 2006, Müller et al., 2007] is another object-centric approach to process implementation, which has been developed specifically to support processes in the automotive industry. The key requirement for automotive processes that deal with assembly of complex products is the coordination of all the subprocesses that need to be carried out as part of the overall assembly. Müller et al point out that the required coordination of sub-processes has a strong correlation to the dependencies between product parts that the sub-processes manipulate. Taking into account this central role of parts[1] or more generally objects, this approach is founded on modeling of object structural interrelations as data models and dynamic behavior for each object type as object life cycle components.

Each state transition in an object life cycle component, called an internal state transition, is associated with a sub-process manipulating objects of this type. Therefore, an object life cycle component represents coordination of sub-processes that deal with one object type. Coordination of sub-processes that deal with different object types, i.e. synchronization of object life cycle components, is achieved by external state transitions, which connect states in different object life cycle components. Each relation between object types in the data model gives rise to an external state transition between the object life cycle components for these object types. The object life cycle components together with their external state transitions form a Life Cycle Coordination Model (LCM).

Other approaches to process implementation that can be considered object-centric or are closely related to object-centric approaches include that adopted in the FlowConnect system [Redding et al., 2007], case handling [van der Aalst et al., 2005], document-driven workflows [Wang and Kumar, 2005] and product-based workflow design [Reijers et al., 2003].

[1] These are referred to as "components" in the original publications on data-driven process structures.

7.1.2 Advantages and Challenges of Object-Centric Approaches

Potential advantages of using an object-centric approach to process implementation as opposed to an activity-centric one include explicit management of object states [Nandi and Kumaran, 2005], a higher degree of reuse [van der Aalst et al., 2001] and flexibility [Müller et al., 2007]. Furthermore, object-centric implementations can be used for distributed process execution and can lead to a more maintainable and adaptable implementation than activity-centric approaches, since the behavior of one component can be partially changed without influencing the rest of the components [Müller et al., 2006].

However, the more synchronizations there are between the object life cycle components, the costlier becomes their distribution and the more complicated it is to change and reuse their behavior. This is illustrated in Figure 7.1, where the components in (b) require more synchronizations than the components in (a) and are therefore more difficult to distribute, change and reuse. One of the challenges in object-centric process implementation is therefore the management of component interdependencies, commonly referred to as *coupling* in software engineering [Briand et al., 1999].

An intuitive approach to deriving object life cycle components from a process model that specifies the process logic to be implemented is to produce a component for each object type used in the process model [Bhattacharya et al., 2005]. However, if such an approach does not explicitly address object life cycle interdependencies, it runs the risk of producing components that are highly coupled. Component revision, e.g. moving some behavior from one component to another or merging components, is one approach to reducing coupling, illustrated in Figure 7.2(a). However, as a result the process model can get out of sync with its implementation, which challenges the propagation of any subsequent process model changes to the implementation. This problem can be alleviated by making the modeler aware of the expected coupling before component derivation, so that the process model can be adapted until a desired level of coupling is achieved. As illustrated in Figure 7.2(b), realization of this approach requires the computation of component synchronization and the expected component coupling based on a given process model.

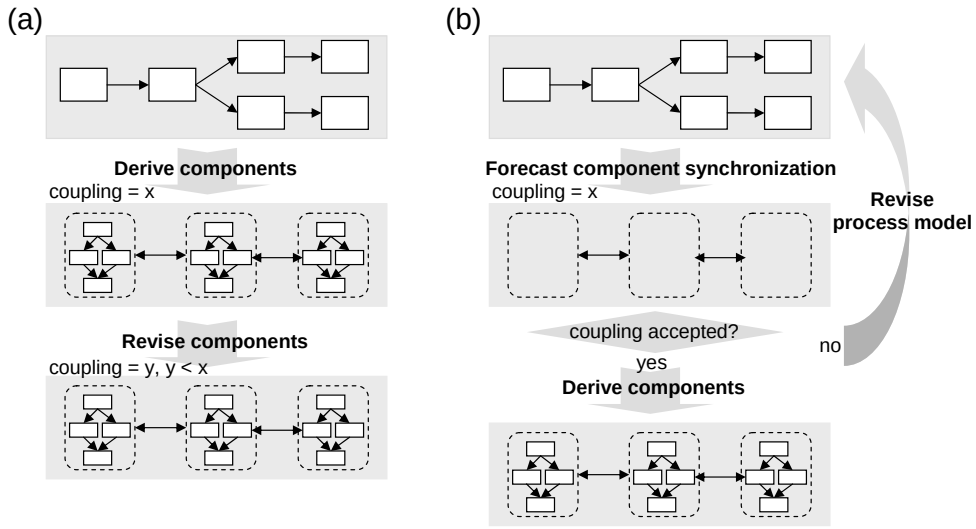


Figure 7.2: Two approaches to alleviating coupling

The derivation of object life cycle components from a process model is not described in detail in the existing literature. In [Bhattacharya et al., 2005], one example of mapping

a process model to ABOs is provided without a detailed description of the mapping. In a follow-up work by Kumaran et al [Kumaran et al., 2008], an algorithm for generating object life cycle components with synchronized transitions from a process model is presented. However, the algorithm does not ensure that the control-flow semantics of the original process model is preserved in the generated object life cycle components. Similarly, our object life cycle extraction transformation presented in Chapter 6 does not suffice for the purpose of object life cycle component derivation, since it does not necessarily preserve all the control-flow logic of a process model.

Therefore, in this chapter, we first investigate how a complete decomposition of control flow in a process model among several object life cycle components can be achieved by demonstrating the mapping of the most common workflow patterns [van der Aalst et al., 2003, Russell et al., 2006a]. Based on this investigation, we identify process model constructs that introduce synchronizations between object life cycle components and therefore contribute to the component coupling. We then show that given a process model, it is possible to compute the component pairs that require synchronization by analyzing the control flow between tasks that change the state of objects. Finally, we use this information to compute the expected coupling of the object life cycle components, necessary for the realization of the approach shown in Figure 7.2(b).

We use Business State Machines (BSMs) [Beers and Carey, 2006], a service orchestration language offered as an alternative to BPEL in IBM WebSphere Integration Developer[2], to represent object life cycle components. We choose to use BSMs here, since it is a language that is already supported by a mature commercial product and could be considered analogous to BPEL, which is the standard for activity-centric process implementation. At the end of the chapter, we discuss the generalization of our approach.

7.2 Business State Machines and Interface Coupling

A BSM is a finite state automaton, tailored for the execution in a service-oriented environment. An example of two BSMs is shown in Figure 7.3. Each BSM can have several of the following: *interfaces*, *references* and *variables*. The $Worker$ BSM in Figure 7.3 has two interfaces: $basic$ comprising the $start$ and $stop$ *operations*[3], and $stateQuery$ with the $getState$ operation. $start$, $stop$ and $getState$ are the three operations that can be invoked on the $Worker$ BSM. On the other hand, the $Observer$ BSM has one interface with two operations, $start$ and $stop$. $Observer$ also has one reference w, referencing the $stateQuery$ interface of the $Worker$ BSM. The $Observer$ and $Worker$ BSMs have one variable each: $wState$ and $shift$, initialized to the literal "$Unknown$" and 0, respectively.

State transitions in BSMs follow the *event-condition-action* paradigm. A transition can be triggered either by an expiration of a *timeout* or by an invocation of an operation defined in one of the BSM's interfaces. Once a transition has been triggered, its associated *condition*, if any, is evaluated. If the condition evaluates to true or there is no condition, the *action* associated with the transition, if any, is performed and the target state of the transition is entered. An action either invokes an operation on one of the BSM's references or performs some other processing specified in a custom language, such as Java.

For example, once the $Observer$ BSM is in state $ready$, a self-transition is triggered repeatedly after the expiration of timeout $wait$. Each time the transition is triggered and

[2] `http://www.ibm.com/software/integration/wid`

[3] In BSMs, operations also have parameters, which we omit here for conciseness.

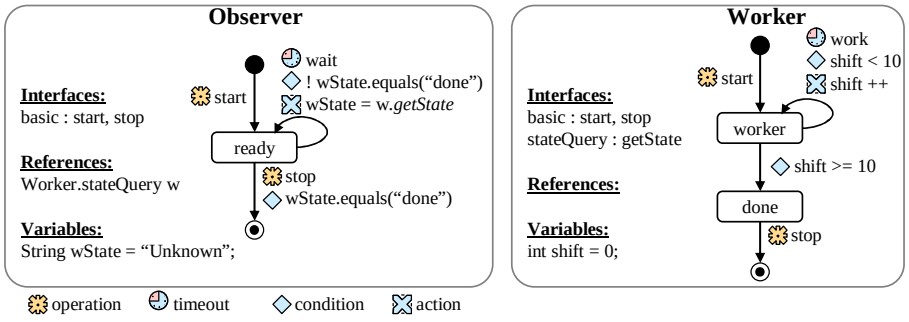


Figure 7.3: Example BSM

$wState$ is not equal to "$done$", the $getState$ operation is invoked on w (invocation is indicated using italics in the diagrams). The operation $getState$ is implicitly handled by every BSM and returns the BSM's current state. The invocation of the $stop$ operation on $Observer$ results in a transition to the final state only if $wState$ is equal to "$done$".

At run time, each *BSM instance* is associated with a *correlation ID*. The run-time engine creates a new BSM instance if it receives a call to an operation associated with an initial transition of some BSM and this operation call specifies a correlation ID that does not correspond to an existing BSM instance. For example, the $Observer$ and $Worker$ BSMs can be instantiated by invoking their $start$ operation with fresh correlation IDs. Furthermore, the correlation ID of the $Worker$ BSM instance needs to be passed to the $Observer$ BSM on instantiation as a parameter of the $start$ operation, to enable the $Observer$ BSM instance to communicate with the $Worker$ instance. In this chapter, we do not explicitly represent the exchange of correlation IDs.

When BSMs are used to represent object life cycle components, component synchronization is achieved through operation invocations, which requires interface bindings between BSMs. We use the *Service Component Architecture (SCA)* [SCA, 2007], which is a service-oriented component framework, to represent these bindings. An *assembly model* in SCA is a representation of directed interface bindings, called *wires*, between components. The assembly model for the BSMs from Figure 7.3 is shown in Figure 7.4. Invocation of operations by the $Observer$ BSM on the $Worker$ BSM requires that the components are connected by a wire in the assembly model.

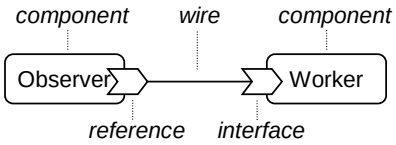


Figure 7.4: Assembly model

Given an assembly model, we use the term coupling to refer to the interdependencies of the contained components. We quantify the coupling of an assembly model by defining the *interface coupling* metric, adapted from existing work on quality metrics in the business process domain [Reijers and Vanderfeesten, 2004]. In the following, we introduce a definition of an assembly model, followed by the definition of interface coupling.

Definition 79 (Assembly model)**.** *An* assembly model *is a tuple $M = (C, \phi)$, where C is the set of components in M and $\phi \subseteq C \times C$ is the wire relation between components.*

Definition 80 (Interface coupling)**.** *Given an assembly model* $M = (C, \phi)$*, its* interface coupling *is defined as follows:*

$$p(M) = \begin{cases} 0 & \text{if } |C| = 0 \text{ or } 1 \\ \frac{|\phi|}{|C| \times (|C|-1)} & \text{otherwise} \end{cases} \tag{7.1}$$

Interface coupling represents the ratio between the actual number of wires and the maximum possible number of wires between the components in an assembly model. A coupling value of 0 means that there is no interaction at all between the components. This implies that the distribution of these components does not incur any communication costs, and the implementation of each component can be maintained and its behavior adapted at run time without side-effects on the other components. On the contrary, a coupling value of 1 means that every component interacts with every other component. The distribution of such components would incur high communication costs, and maintenance or adaptation of one component would affect the behavior of all other components.

For example, the interface coupling of the assembly model shown in Figure 7.4 is $\frac{1}{2\times1} = 0.5$. The assessment of such an interface coupling value requires a certain threshold be set, above which coupling values should be considered high. Such a threshold can be evolved as a best practice by modelers and developers, i.e. first initialized to some value and then refined on further iterations or projects based on the experience gained in deploying and maintaining object-centric implementations. Empirical evaluations can also help in determining a generic guideline for this threshold.

More refined coupling metrics could also be used here, e.g. to take into account the number of operations in the component interfaces connected to wires or the number of operation calls inside the BSMs.

In the following section, we investigate the mapping of process models to BSMs on the basis of worklfow patterns. We identify process model constructs that lead to the synchronization of BSMs, hence introducing wires into the resulting assembly model and contributing to the overall interface coupling.

7.3 Mapping Process Models to Business State Machines

Workflow patterns [van der Aalst et al., 2003, Russell et al., 2006a] are a well-established benchmark for exploring how common process behaviors can be represented in different process modeling and implementation languages. In this section, we show how the basic control-flow patterns, WP1-WP5, can be mapped to object life cycle components represented by BSMs. These control-flow patterns can be expressed by the workflow graphs that we use to represent process models in this dissertation.

We provide BSM mappings on an exemplary basis, similar to what has been done in the existing language evaluations based on workflow patterns (e.g. [Wohed et al., 2003]). Furthermore, we discuss the requirements that a generic instance of each workflow pattern has with respect to the synchronization of BSMs and the overall interface coupling.

The main goal of the mapping is to distribute the actions in a given process model among several object life cycle components represented by BSMs, while preserving the overall control-flow logic. A separate component is created for a particular object type if objects of this type are created by actions in the given process model or if the process model induces transition for objects of this type. In the following, we make a simplifying assumption that one action creates or changes the state of objects of exactly one object type, which we call the *control object type* for that action. Each action is then placed into

exactly one object life cycle component created for the control object type of the action. We also assume that the outgoing edges of one decision constrain the states of objects of exactly one object type, which we call the control object type for that decision. In Section 7.5, we explain how our approach can be extended to relax these simplifying assumptions.

7.3.1 WP1 Sequence

Several actions[4] are executed one after another in the WP1 sequence pattern, as illustrated with two examples in Figure 7.5. In this chapter, we abstract from the type of data flow used in process models and only focus on the induced transitions for different object types. We use annotations in the form $t[s_1 \rightarrow s_2]$ to indicate that $(a, s_1, s_2) \in induced(t)$ for a given action a (cf. Section4.3.3 of Chapter 4). To indicate that an action creates objects of a given type in a particular state, the source state in the annotation is omitted: $t[\rightarrow s_1]$.

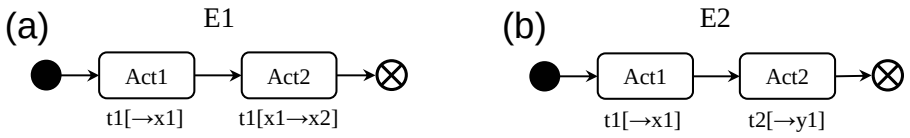


Figure 7.5: WP1 examples

In E1 shown in Figure 7.5, $Act1$ and $Act2$ change the state of objects of the same object type, $t1$. The mapping of E1 is straightforward, as shown in Figure 7.6(a). The actions $Act1$ and $Act2$ are mapped to actions associated with state transitions in the BSM. The assumption is that there exists one *coordinating component*, which manages the instantiation and halting of BSMs by invoking the $start$ and $stop$ operations. Once the $t1$ BSM shown in Figure 7.6(a) is instantiated by the coordinating component, it enters state $Idle$. Actions $Act1$ and $Act2$ are then executed in a sequence, changing the state of the $t1$ BSM instance first to $x1$ and then to $x2$. The $t1$ BSM instance reaches the final state and halts once the coordinating component invokes its $stop$ operation. For conciseness, interfaces and references are omitted in the presented BSM diagrams.

In E2 shown in Figure 7.5(b), $Act1$ and $Act2$ change the state of objects of different object types, $t1$ and $t2$. The mapping of E2 is shown in Figure 7.6(b), where BSMs $t1$ and $t2$ represent the object life cycles for object types $t1$ and $t2$, respectively. The coordinating component instantiates both BSMs, taking each one to the $Idle$ state. Once the $t1$ BSM instance executes $Act1$, it transits to state $Notifying\ t2$. In this state, the $t1$ BSM instance queries the state of the $t2$ BSM instance repeatedly. Once the $t2$ BSM instance has reached the $Idle$ state, the $t1$ BSM instance notifies the $t2$ BSM instance that it has reached the state $x1$ by invoking the $t1x1$ operation. After this, the $t1$ BSM instance transits back to state $x1$, and the $t2$ BSM instance executes the $Act2$ action.

The E1 mapping comprises one component, as shown in the assembly model in Figure 7.6(c). Since there are no wires in this assembly model, the interface coupling is 0. The E2 mapping comprises two components with one wire between them, as shown in Figure 7.6(d). The interface coupling is $\frac{1}{2\times1} = 0.5$.

[4]We use the term "action" to be consistent with the remainder of the dissertation, although the workflow pattern literature generally uses the term "activity".

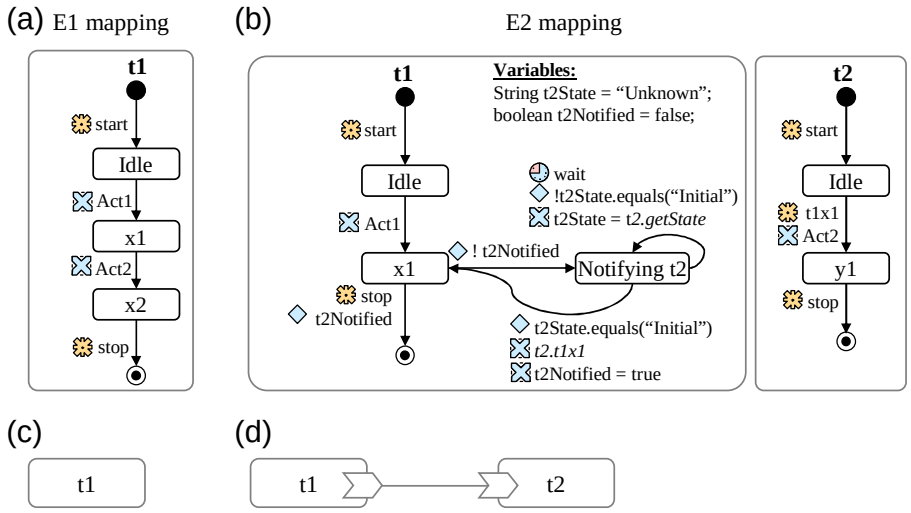


Figure 7.6: WP1 example mappings

WP1 Synchronization Requirements

A generic instance of WP1 comprises actions $a_1, ..., a_n$ that change the states of objects of object types $t_1, ..., t_n$, respectively. A pair of actions a_i, a_{i+1}, with $1 \leq i < n$, requires a synchronization of BSM instances t_i and t_{i+1} if and only if $t_i \neq t_{i+1}$. Such synchronizations represent the handover of control between BSM instances, therefore we refer to them as *control handover* synchronizations or simply control handovers. Each such control handover requires a wire from the component t_i to the component t_{i+1} to be present in the assembly model. The introduction of these wires contributes to the overall interface coupling of the resulting assembly model.

7.3.2 WP2 Parallel Split & WP3 Synchronization

In the WP2 parallel split pattern, several actions are executed simultaneously or in any possible order. In the WP3 synchronization pattern, several parallel threads are joined together into a single control thread. An example containing an instance of both of these workflow patterns is shown in Figure 7.7. Note that we do not only consider block-structured process models, but examine these two patterns together for the sake of conciseness. In E3, each action changes the state of an object of a different object type.

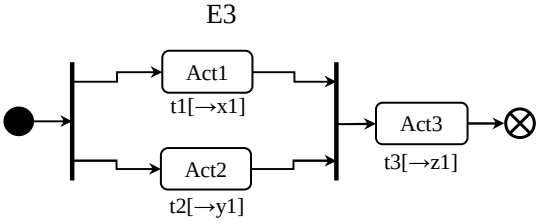


Figure 7.7: WP2 & WP3 example

The mapping of E3 is shown in Figure 7.7. Since all BSM instances are by default executed concurrently, no explicit representation of the parallel split is required. Synchronization of the threads is performed using notifications, similarly to the way it is done in the E2 mapping in Figure 7.6(b). The $t3$ BSM instance waits to receive notifications from

the $t1$ and $t2$ BSM instances, i.e. invocations of operations $t1x1$ and $t2y1$, before executing $Act3$.

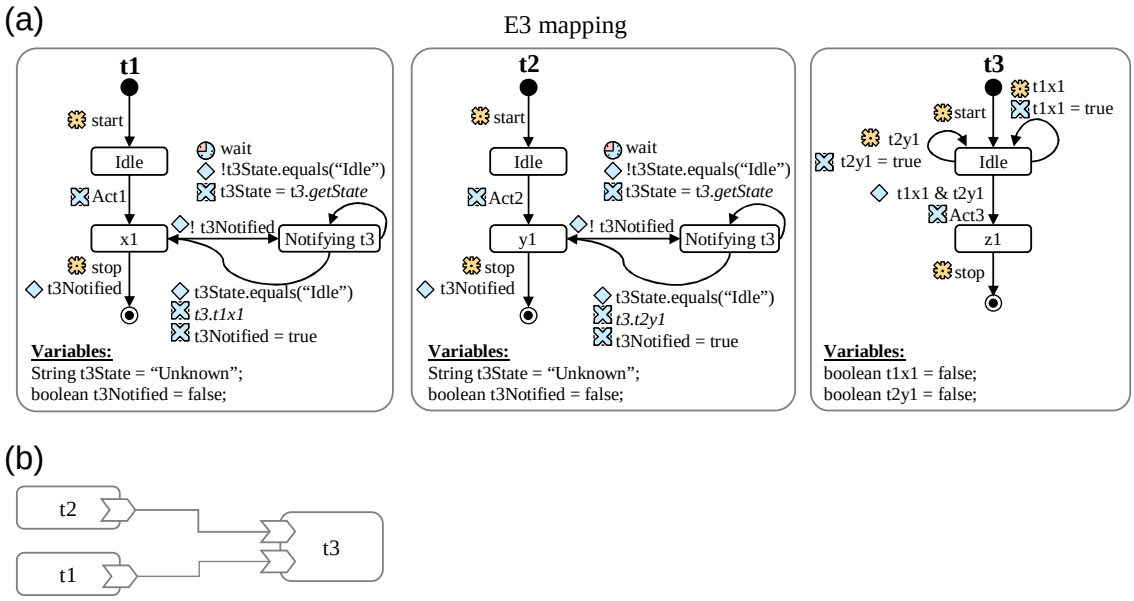


Figure 7.8: WP3 & WP3 example mapping

The assembly model for the E3 mapping comprises three components and two wires, as shown in Figure 7.8(b). The interface coupling of this assembly model is $\frac{2}{3\times2} \approx 0.33$.

WP2 & WP3 Synchronization Requirements

As instances of WP2 do not require any interaction between BSM instances, they do not contribute wires to the assembly model and have no effect on the interface coupling.

A generic instance of WP3 comprises actions $a_1, ..., a_n$ that all need to complete before action a_{n+1} can begin execution. Assuming that $a_1, ..., a_n, a_{n+1}$ respectively change the states of objects of object types $t_1, ..., t_n, t_{n+1}$, a pair of actions a_i, a_{n+1}, with $1 \leq i \leq n$, requires a synchronization of BSM instances if and only if $t_i \neq t_{n+1}$. Such synchronizations are control handovers, introduced in Section 7.3.1, since the purpose of these syncrchronizations is to hand over control between BSM instances.

7.3.3 WP4 Exclusive Choice & WP5 Simple Merge

In the WP4 exclusive choice pattern, one out of several actions is executed based on the outcome of a decision. In the WP5 simple merge pattern, several alternative threads are joined into one control thread without synchronization. An example containing instances of these patterns is shown in Figure 7.9. The annotation $t1[\rightarrow x1, x2]$ indicates that $Act1$ creates an object of type $t1$ either in state $x1$ or $x2$.

The mapping of E4 is shown in Figure 7.10(a). Once an instance of the $t1$ BSM is created by the coordinating component, the $t1$ BSM instance executes $Act1$ and transits to the intermediate state $x1x2$. The state of the $t1$ BSM instance is then updated either to $x1$ or $x2$ depending on the outcome of the $Act1$ execution. The evaluation of the decision in E4 is implicitly performed by the $t1$ BSM instance: if its current state is $x1$, the $t1$ BSM instance executes $Act2$, otherwise if its current state is $x2$, it notifies the $t2$ BSM instance

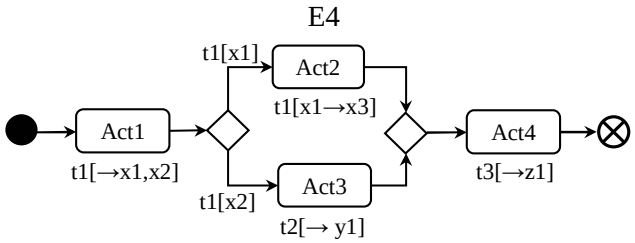


Figure 7.9: WP4 & WP5 example

by invoking the $t1x2$ operation. The $t2$ BSM instance executes $Act3$ only if it receives an invocation of $t1x2$, and does nothing otherwise.

The merging of alternative control threads is implemented similarly to the synchronization in Figure 7.8, except that the $t3$ BSM instance executes $Act4$ as soon as it receives an invocation of *either* operation $t1x3$ or operation $t2y1$.

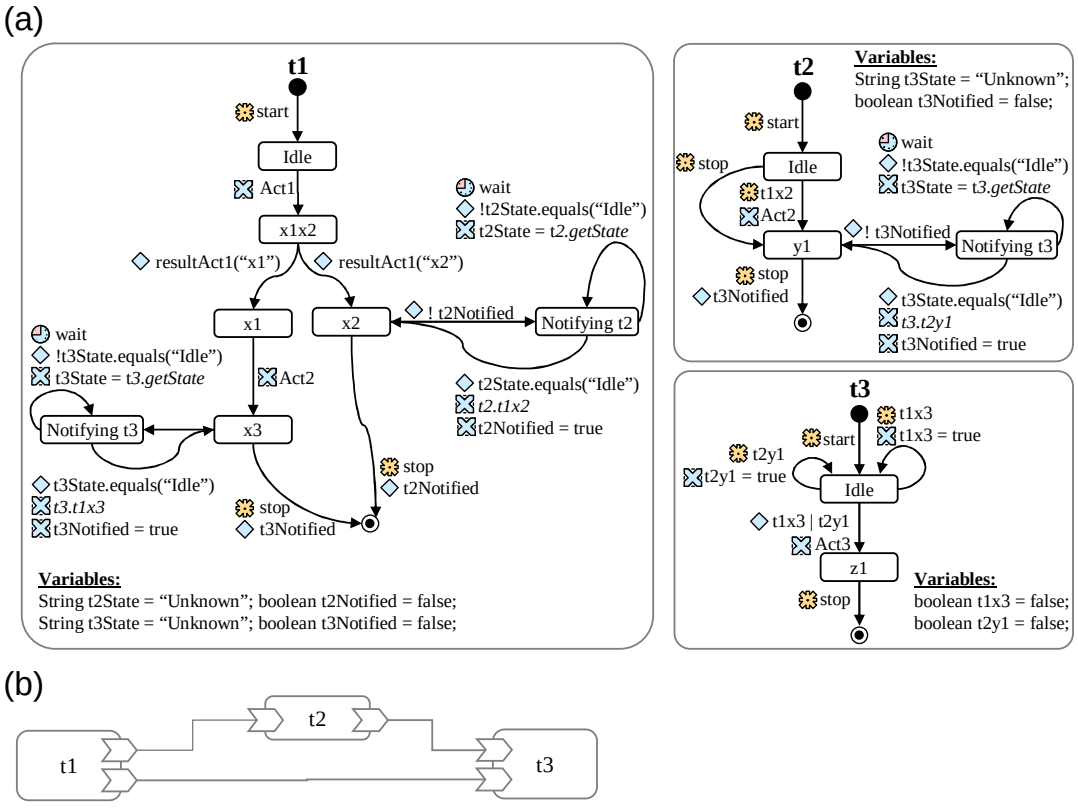


Figure 7.10: WP4 & WP5 example mapping

The assembly model of the E4 mapping comprises three components and three wires, as shown in Figure 7.10(b). The interface coupling of this assembly model is $\frac{3}{3\times2} = 0.5$. These three components have a higher coupling value than those in Figure 7.8(b), because of an additional wire between $t1$ and $t2$ required for communicating the decision outcome.

WP4 & WP5 Synchronization Requirements

A generic instance of WP4 comprises an action a with a control object type t, followed by a decision d with a control object type t_d, followed by actions $a_1, ..., a_n$ with control

object types $t_1, ..., t_n$. If $t \neq t_d$, then an instance of BSM t requires to hand over control to an instance of BSM t_d, so that the t_d BSM instance can influence the subsequent execution according to its state. Depending on the state of the t_d BSM instance, one of the $a_1, ..., a_n$ actions is executed. The t_d BSM instance thus requires synchronization with BSM instances t_i, where $1 \leq i \leq n$ and $t_d \neq t_i$. All these are control handover synchronizations.

A generic instance of WP5 comprises actions $a_1, ..., a_n$, one of which needs to complete before action a_{n+1} can begin execution. Assuming that $a_1, ..., a_n, a_{n+1}$ respectively change the states of objects of object types $t_1, ..., t_n, t_{n+1}$, a pair of actions a_i, a_{n+1}, with $1 \leq i \leq n$, requires a synchronization of BSM instances if and only if $t_i \neq t_{n+1}$. These are also control handover syncrhonizations.

In this section, we have demonstrated how instances of workflow patterns WP1-WP5 can be mapped to object life cycle components represented by BSMs. Another workflow pattern commonly encountered in process models is WP10 arbitrary cycles. This pattern can be implemented in BSMs as a combination of WP5 and WP4 pattern instances. Similar to instances of WP5 and WP4 patterns, instances of the WP10 pattern require control handover synchronizations.

In terms of control flow, the expressivity of workflow graphs used in this dissertation to represent process models is described exactly by the workflow patterns WP1-WP5, WP10 and WP11 implicit termination, since a workflow graph cannot be used to model instances of any other control-flow patterns. The WP11 pattern requires that the process execution terminates when there are no activated nodes remaining. In the mapping to BSMs, we assume that the coordination component halts all BSMs when no other transitions inside BSMs can be triggered. Hence, the mapping described in this section is sufficient to map workflow graphs to BSMs. The interested reader is also referred to [Wahler and Küster, 2008], where we show how the processing of object collections with the WP14 multiple instances with a priori run-time knowledge (not expressible in a workflow graph) can be mapped to BSMs.

In the following, we demonstrate the mapping based on a concrete process model example comprising instances of several workflow patterns.

7.3.4 Mapping an Example Process Model

As an illustrative example, we use a process designed for the organization of alumni events at the IBM Zurich Research Laboratory. An abridged process model for this process is shown in Figure 7.11.

After the approval of the budget, the date for the event is fixed and then two things happen in parallel: the program, invitations and web site are prepared; and catering is organized. After all these have completed, the alumni event is hosted. The process model contains two subprocesses: $Fix\ Dates$ and $Develop\ Web\ Site$. Subprocesses are shown in Figure 7.11 using the UML Activity Diagram notation for StructuredActivityNodes (cf. Chapter 2). Note that subprocess hierarchy is flattened in the workflow graph representation of such a process model.

Action annotations indicate induced transitions and first states for different object types used in this process model. For example, the $Create\ Web\ Site$ action creates an object of type $Web\ Site$ in state $Draft$, and $Publish\ Web\ Site$ changes the state of the $Web\ Site$ object from $Draft$ to $Published$. Additionally, last states are underlined in the action annotations.

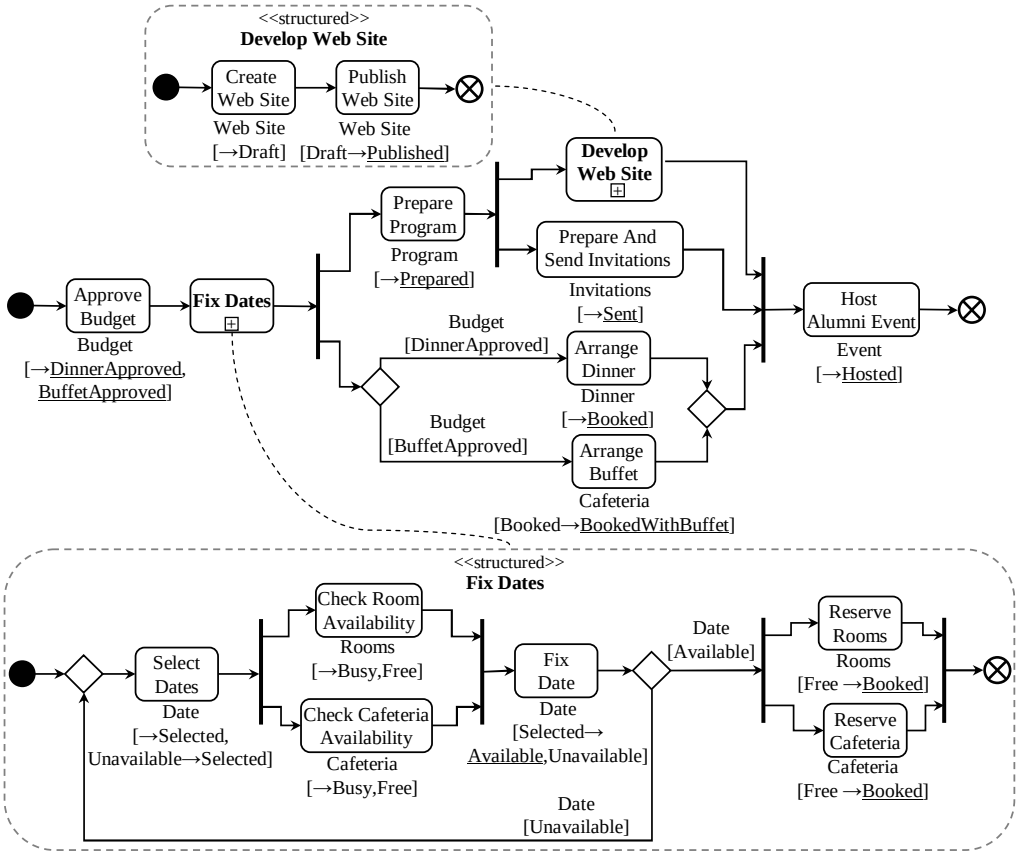


Figure 7.11: Process model for alumni event organization

In an object-centric implementation of the alumni event organization process, the process logic is split into nine object life cycle components ($Budget$, $Cafeteria$, $Date$, $WebSite$, etc.). Actions that change the state of objects of type $Web\ Site$, namely $Create\ Web\ Site$ and $Publish\ Web\ Site$, participate in instances of workflow patterns WP1 sequence, WP2 parallel split and WP3 synchronization. Assuming the presented mappings for different workflow patterns, the BSM shown in Figure 7.12 is produced for the $Web\ Site$ object type. By examining the $WebSite$ BSM, we can determine that it requires synchronization with the $Program$ and $Event$ BSMs.

All the required synchronizations between different BSMs and the overall interface coupling are not trivial to determine by considering the original process model, which comprises instances of different workflow patterns and changes the states of objects of many object types. However, we wish to obtain such information without having to generate the actual BSMs, in order to support the approach to tackling high coupling shown in Figure 7.2(b). In the following section, we describe how the expected interface coupling can be computed directly based on a given process model.

7.4 Computing Expected Interface Coupling

Given a process model, the number of components in the assembly model of its BSM implementation is equal to the number of distinct control object types for the actions and decisions in that process model. In Section 7.3, we showed that the number of wires between the components depends on the control handover synchronizations be-

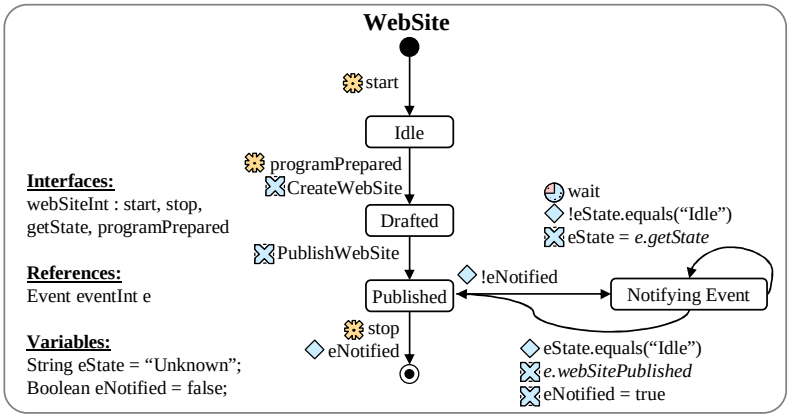


Figure 7.12: BSM for Web Site object type

tween the BSMs. Since all the synchronizations required by different patterns are control handover synchronizations, we directly compute all object type pairs that require such synchronizations, instead of first identifying workflow pattern instances in a given process model.

A control handover is required whenever a given process model contains a path p from an action or a decision n_1 with a control object type t_1 to another action or decision n_2 with a different control object type t_2 such that there are no other actions or decisions on the path p. To compute the object types that require control handovers, we propagate the information about control object types downstream and upstream from each action and decision.

In the following, we first give a formal definition of control object types.

Definition 81 (Control object types)**.** *Let a workflow graph $G = (N, E)$ with repository or routed data flow using a set of object types T and a state specification be given. We define a function $cot : N \to \mathcal{P}(T)$ to map a node to a set of* control object types*. Given a node $n \in N$, $cot(n)$ is defined as follows:*

$$cot(n) = \begin{cases} \{t \in T \mid \exists s_1, s_2 \in S_t.\ (n, s_1, s_2) \in induced(t) \vee & \text{if } n \text{ is an action} \\ t \in dataout(n) \setminus datain(n)\} & \\ \{t \in T \mid \exists e \in out(n).\ cond(e, t) \neq S_t\} & \text{if } n \text{ is a decision} \\ \emptyset & \text{otherwise} \end{cases}$$

Downstream and *upstream control object types* are now defined for storing propagated information about control object types on the edges of a given process model.

Definition 82 (Downstream and upstream control object types)**.** *Let a workflow graph $G = (N, E)$ with repository or routed data flow using a set of object types T be given. We define functions $dcot, ucot : E \to \mathcal{P}(T)$ to map each edge $e \in E$ to a set of* downstream *and* upstream control object types*. Given an edge $e \in E$, $dcot(e)$ and $ucot(e)$ are defined as follows:*

$$dcot(e) = \begin{cases} \emptyset & \text{if } e \in out(n) \text{ and } n \text{ is the start node} \\ cot(n) & \text{if } e \in out(n) \text{ and } n \text{ is an action or a decision} \\ \bigcup_{e' \in in(n)} dco(e') & \text{if } e \in out(n) \text{ and } n \text{ is not the start node,} \\ & \text{an action or a decision} \end{cases}$$

$$ucot(e) = \begin{cases} \emptyset & \textit{if } e \in in(n) \textit{ and } n \textit{ is the stop node} \\ cot(n) & \textit{if } e \in in(n) \textit{ and } n \textit{ is an action or a decision} \\ \bigcup_{e' \in out(n)} uco(e') & e \in in(n) \textit{ and } n \textit{ is not the stop node,} \\ & \textit{an action or a decision} \end{cases}$$

Downstream and upstream control object types can be computed for a given process model using iterative data-flow analysis [Kam and Ullman, 1976], similarly to the way effective input and output states were computed in Section 4.4 of Chapter 4. For example, to compute the downstream control object types, $dcot(e)$ is initialized to an empty set for each edge e and then the nodes in the process model are traversed, evaluating the $dcot$ equations (Definition 82) for each outgoing edge of the traversed node. Reverse postorder traversal ensures that in the absence of cycles each node is visited once. In the presence of cycles, the nodes are traversed repeatedly until a fixpoint is reached, i.e. an iteration when no $dcot$ values are updated.

Figure 7.13 shows the alumni event organization process model with the downstream and upstream control object types indicated above and below each edge, respectively. Action and object type names are abbreviated for conciseness, and decisions are labeled with their control object types. For example, for edge e connecting the start node and the AB action, $dcot(e) = \{\}$ since the start node has no control object types, and $ucot(e) = \{B\}$ since B is the control object type of action AB. For the outgoing edges of actions AD and $AB2$, e_1 and e_2 respectively, $dcot(e_1) = \{N\}$ and $dcot(e_2) = \{C\}$. The union of these object types produces the downstream control object types for the edge e_3 going out of the merge following the actions AD and $AB2$: $dcot(e_3) = \{N, C\}$.

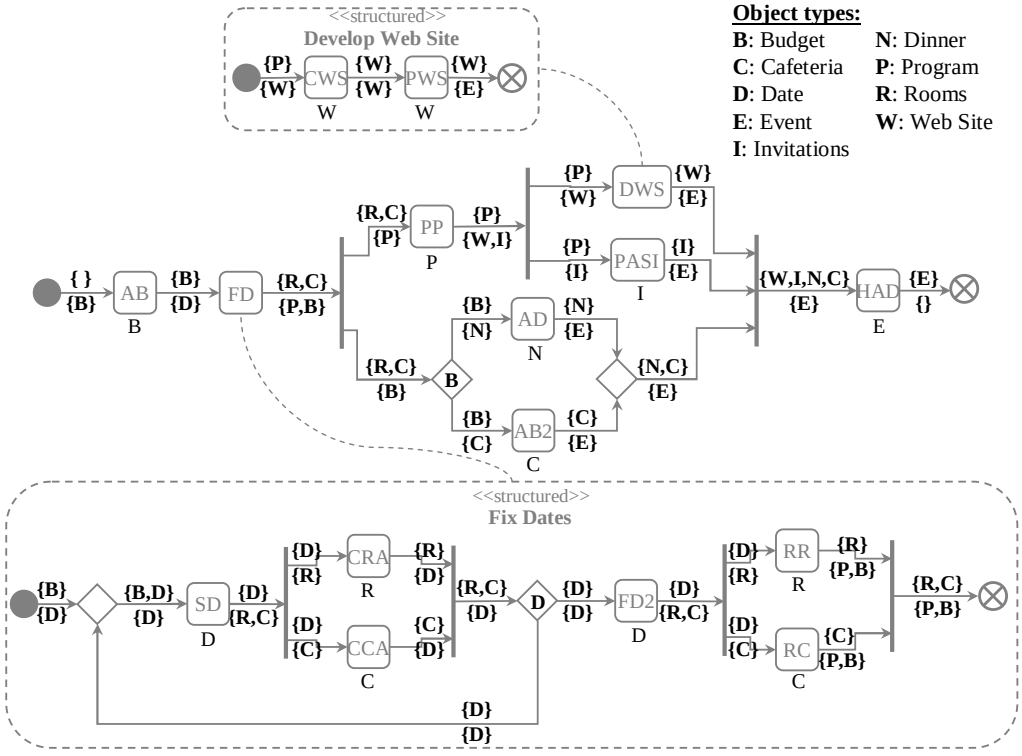


Figure 7.13: Downstream (above edges) and upstream (below edges) control object types

The set of object type pairs that need to perform control handovers is then defined as follows.

Definition 83 (Control handover object type pairs)**.** *Let a workflow graph $G = (N, E)$ with repository or routed data flow using a set of object types T be given. We define the function $chop : N \to \mathcal{P}(T \times T)$ to map each node to a set of* control handover object type pairs. *Given a node $n \in N$, $chop(n)$ is defined as follows:*

$$chop(n) = \begin{cases} \bigcup_{e \in in(n)} (dcot(e) \times ucot(e)) \ \setminus \ \{(t,t) \in T \times T\} & \text{if } n \text{ is an action or a decision} \\ \emptyset & \text{otherwise} \end{cases}$$

For example, the incoming edge of the decision with control object type B gives rise to two control handover object type pairs: (R, B) and (C, B). On the other hand, the incoming edge of the AD action gives rise to only one control handover object type pair: (B, N).

The *forecasted assembly model* for a BSM implementation of a given process model can now be constructed by introducing a component for each distinct control object type and a wire for each of the control handover object type pairs.

Definition 84 (Forecasted assembly model)**.** *Let a workflow graph $G = (N, E)$ with repository or routed data flow using a set of object types T be given. The* forecasted assembly model *for G is defined as follows:*

$$M = (C, \phi), \text{ where:}$$

- $C = \{c_{t_1}, ..., c_{t_m}\}$ *is the set of components, with one component c_{t_i} for each object type $t_i \in T$ where $1 \leq i \leq m$ and $t_i \in cot(n)$ for some node $n \in N$;*
- $\phi = \{(c_{t_1}, c_{t_2}) \in C \times C \mid (t_1, t_2) \in \bigcup_{n \in N} chop(n)\}$ *is the wire relation between components.*

Given a forecasted assembly model, interface coupling can be directly computed according to Definition 80. The forecasted assembly model for the alumni event organization process model is shown in Figure 7.14. The interface coupling is computed for the entire assembly model and for all component subsets according to Definition 80. In Figure 7.14, the overall interface coupling is $\frac{17}{9 \times 8} \approx 0.236$, which would not give a reason for concern, assuming for example a threshold of 0.6 (cf. Section 7.2 for setting a threshold). However, component sets $\{D, R\}$, $\{D, C\}$, $\{B, C\}$, $\{B, C, D\}$, $\{B, D, R\}$ and $\{C, D, R\}$ have a coupling value higher than 0.6 and should thus be brought to the attention of the modeler, as shown in Figure 7.14.

Once the expected coupling is forecasted using the proposed approach, the modeler should decide how to deal with each set of highly-coupled components. High coupling may be tolerated for components that have a stable design and do not require distributed deployment. Otherwise, the process model should be revised in such a way that the expected interface coupling between components is reduced.

In the alumni event organization process, the decision leading to actions $Arrange\ Dinner$ and $Arrange\ Buffet$ could take place directly after the $Reserve\ Cafeteria$ action, without waiting for the $Reserve\ Rooms$ action to complete (see Figure 7.15(a)). A possible revision of the process model is shown in Figure 7.15(b), where the $Reserve\ Rooms$ action is placed after the fork and before the $Prepare\ Program$ action. This revision removes the wire between components R and B and the wire between components C and

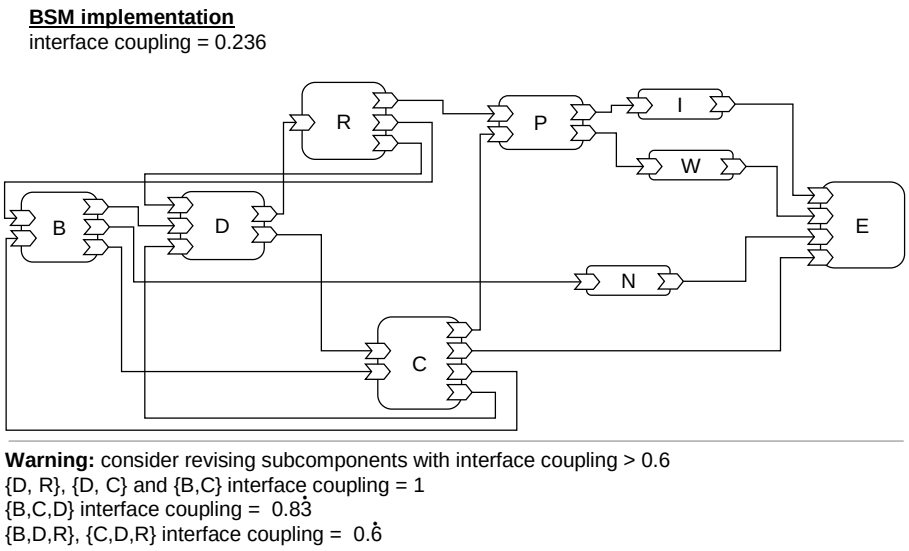


Figure 7.14: Forecasted assembly model for the alumni event organization process model

P. However, a new wire from component C to component R is added. After each process model revision, the coupling computations need to be repeated and results shown to the modeler. As a result of this revision, the overall coupling is reduced to $0.\dot{2}$, the interface coupling of component set $\{B, D, R\}$ is reduced to 0.5 and the interface coupling of the component set $\{C, D, R\}$ is increased to $0.8\dot{3}$.

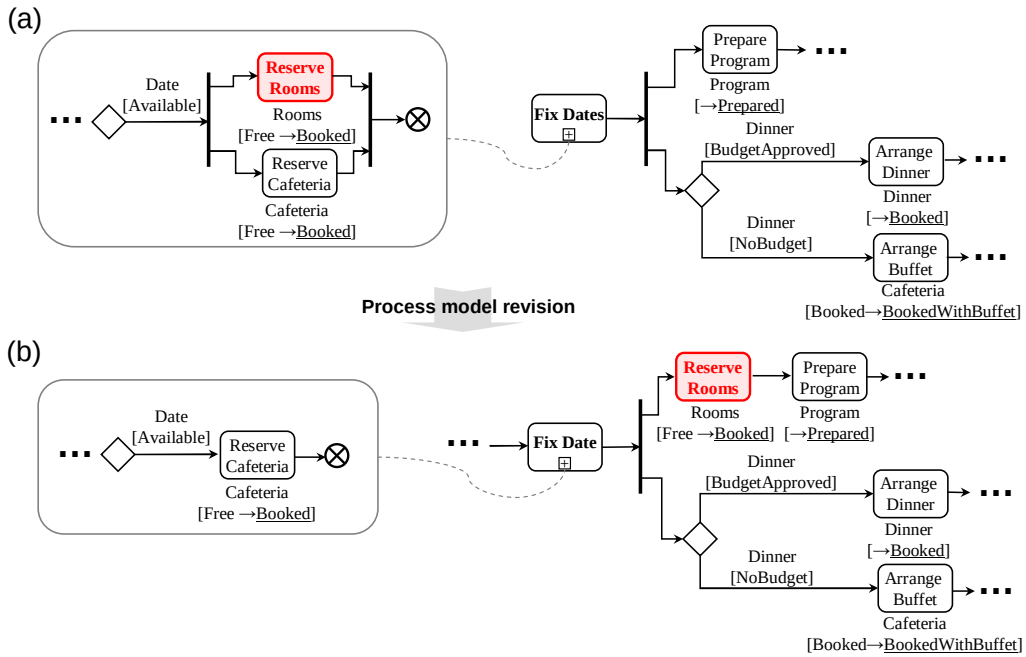


Figure 7.15: Example of a process model revision

Process model revisions should be performed until the modeler is satisfied with the expected interface coupling. At this point, the BSM components themselves can be derived from the process model.

7.5 Generalization of the Approach

In this chapter, we have shown how the expected interface coupling of an object-centric implementation using BSMs can be computed based on a given process model. We assumed that each action in the process model changes the states of objects of one object type. An action that changes the state of several object types would be placed into several BSMs, which would need to synchronize, thus contributing to component coupling. Our current approach can be extended to handle such actions by adding a new synchronization category, *action synchronization*, and providing a definition for computing the object type pairs requiring such a synchronization (similar to Definitions 83).

Our approach can be further extended to handle workflow patterns other than WP1-WP5, WP10 and WP11 by investigating BSM mappings for new patterns, identifying pattern requirements for synchronization of BSMs, and extending Definitions 83 and 84.

Although our approach was demonstrated using BSMs, SCA and the interface coupling metric, it can be generalized to other object-centric approaches, component frameworks (not necessarily based on services) and coupling metrics. For example, ABOs [Nandi and Kumaran, 2005] are based on communicating automata, and our approach is applicable once every ABO has been encapsulated in a component and communication channels between the components have been made explicit. In data-driven process structures [Müller et al., 2007], object life cycles are synchronized by so-called external state transitions. To compute the coupling, each life cycle can be seen as a component, and communication channels need to be introduced between components whose life cycles are connected by external state transitions. Proclets [van der Aalst et al., 2001] use workflow nets to represent object life cycles and make use of explicit communication channels. Although more advanced communication options, such as multicast and broadcast, are supported in proclets, our approach is still applicable.

7.6 Summary and Discussion

In this chapter, we have demonstrated how process models can be mapped to object life cycle components represented by BSMs and presented an approach to tackling component coupling in object-centric process implementations.

The mapping of process models to BSMs was described informally on an exemplary basis. It can serve as a foundation for developing a complete mapping that automates the derivation of BSMs from a given process model. For such a complete mapping, it would be important to show that the behavior of the original process model and the resulting object life cycle components is equivalent. These possible extensions of the presented work lie outside the scope of this dissertation.

Our presented approach to computing the expected component coupling before deriving the actual object life cycle components allows the modeler to alleviate high coupling by directly revising the process model. As opposed to reducing coupling by component refactoring, process model revision before component derivation ensures that the process model and its implementation remain in sync.

We envision that further tool support can greatly assist the modeler during process model revision. A semi-automated approach, similar to the one we described for the resolution of inconsistencies (cf. Chapter 5), could be suitable for guiding the modeler through alternative revisions. The modeler also needs to take into account other effects that revisions have on the resulting components. For example, it would be interesting to determine whether control-flow revisions in a process model can also affect the expected component

complexity and cohesion.

Chapter 8

Tool Support and Method

In this chapter, we describe the implementation of our solution in a prototype and present three modeling strategies that we propose as method support for our solution. In Section 8.1, we provide an overview of how the various components of our solution are implemented in our prototype, called Object Life Cycle Explorer. In Section 8.2, we describe the proposed modeling strategies and explain the scenarios for which each modeling strategy is particularly suitable.

8.1 Object Life Cycle Explorer for WebSphere Business Modeler

We chose to implement our solution as an extension to an existing tool, IBM WebSphere Business Modeler (WBM)[1], which provides state-of-the-art support for business process modeling. This approach allowed us not only to accelerate the implementation, but also to achieve a higher acceptance by target users.

WBM defines a proprietary meta-model and a custom notation for process modeling, which share many aspects with UML Activity Diagrams [UML, 2007b]. Object types are modeled as so-called *business items* (see (1) in Figure 8.1), which can be associated with user-defined attributes. States are considered to be a distinguished attribute of a business item. A separate tab is provided to allow the user to define a set of valid states for a particular business item (see (2) in Figure 8.1). In the process model editor, business items and states can be specified for node inputs/outputs and edges connecting nodes (see (3) in Figure 8.1). Overall, a large subset of process models created with WBM can be represented using the syntax and semantics of process models defined in Chapter 3 of this dissertation.

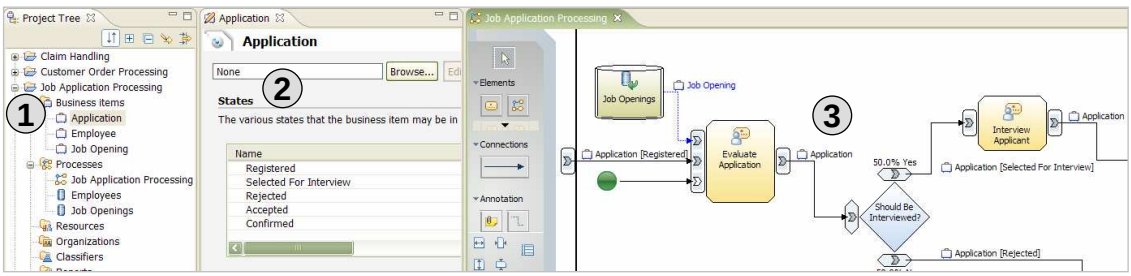
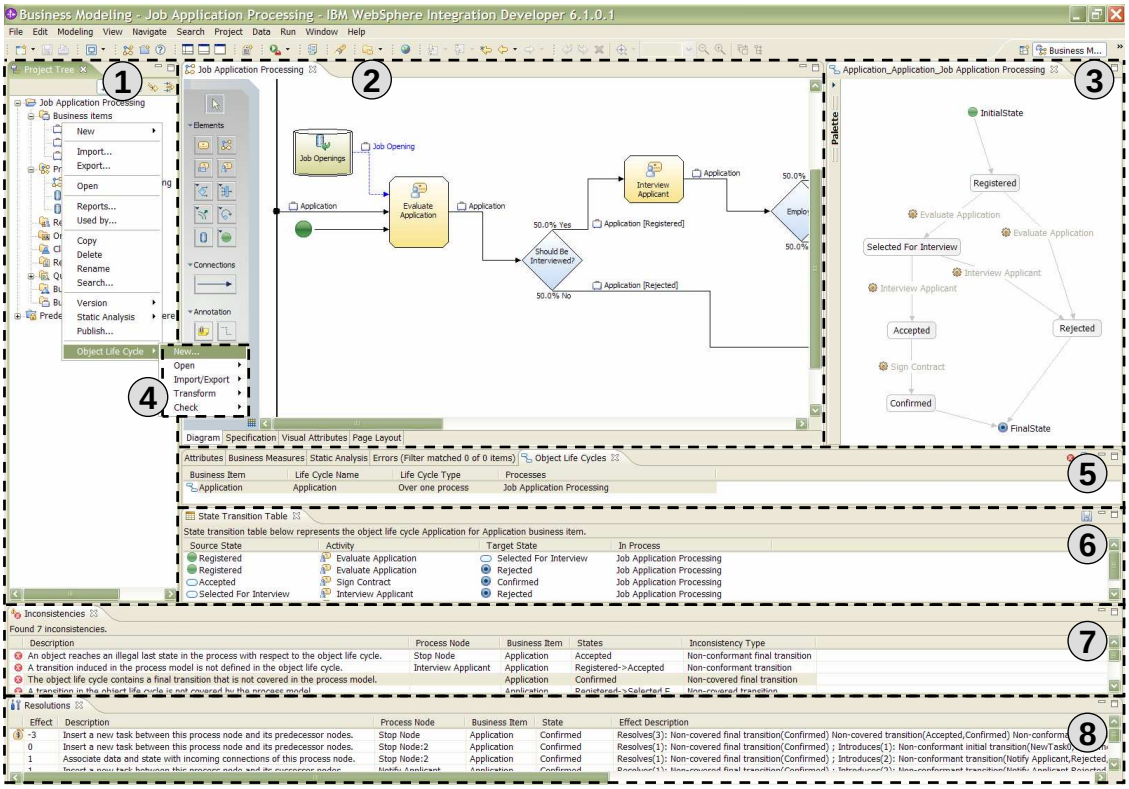


Figure 8.1: Object, state and process modeling in WBM

[1] http://www.ibm.com/software/integration/wbimodeler

Object Life Cycle Explorer extends WBM to support integrated process and object life cycle modeling as described in our solution. WBM is implemented on top of the Eclipse development platform[2], which offers an extension mechanism based on plug-ins [Gamma and Beck, 2003]. Our extension comprises a set of Eclipse plug-ins for WBM, exhibiting itself via several new menus, editors and views, seamlessly integrated into the user interface of WBM. An overview screenshot of Object Life Cycle Explorer is shown in Figure 8.2, with the different parts of the user interface numbered and labeled.

All the features of Object Life Cycle Explorer are available via the object life cycle menu. The modeling and visualization of object life cycle models is facilitated with the object life cycle editor. The state transition table view offers an additional visualization of an object life cycle model, where some additional details about the state transitions are displayed. Object life cycle models are associated with business items and can be managed using the object life cycle menu and view, including object life cycle model creation, deletion, import/export, etc. Consistency checking, inconsistency resolution and model transformations can be invoked via the object life cycle menu. The inconsistencies and resolutions views are used to display the detected inconsistencies and available inconsistency resolutions, respectively.



WBM:

1. project tree
2. process model editor

Object Life Cycle Explorer:

3. object life cycle editor
4. object life cycle menu
5. object life cycles view
6. state transition table view
7. inconsistencies view
8. resolutions view

Figure 8.2: Object Life Cycle Explorer overview screenshot

Table 8.1 gives an account of the aspects of our solution, which are implemented in Object Life Cycle Explorer (denoted by filled squares). As can be seen, only two aspects of the proposed solution were not implemented (denoted by empty squares), namely

[2] `http://www.eclipse.org/platform`

the guided inconsistency resolution and the prediction of object life cycle coupling. These two aspects are however not fundamental to our solution and can easily be added in a complete implementation of the solution.

Object Life Cycle Explorer was released on the IBM alphaWorks[3], which is IBM's portal for emerging technologies, allowing early adopters to experiment with and use prototypes developed at the IBM Research laboratories around the world. Our alphaWorks release [Wahler et al., 2008] provides a download package, which includes extensive tutorials and sample models, apart from the software itself. Table 8.1 shows that most of the implemented features were included in the alphaWorks release. Due to the lack of resources available to perform extensive testing and quality control required for the release, we decided to restrict some of the implemented features to the internal prototype only. To this date, Object Life Cycle Explorer package was downloaded over 100 times from the alphaWorks website and is being used on several client engagements.

Table 8.1: Implemented and released aspects of the proposed solution

Solution aspect	**Implemented**	**Released**
Consistency		
Correctness of state specifications	■	□
Process and object life cycle model consistency	■	■
Inconsistency resolution		
Inconsistency prioritization	□	□
Side-effect forecast	■	■
Cost-based resolution comparison	■	□
Cycle safety analysis	■	□
Model transformations		
Object life cycle extraction	■	■
Process model generation	■	■
Design to implementation transition		
Computation of expected coupling	□	□

In the following, we provide some details about how the consistency checking, inconsistency resolution and model transformations are implemented in Object Life Cycle Explorer. However, first we discuss some specifics about the way WBM currently supports data-flow modeling and state specification, which had an impact on our implementation and the types of models that can be handled.

8.1.1 Data Flow and State Specification in WebSphere Business Modeler

WBM supports the modeling of repository, routed and also mixed data flow. A mixed approach is used in the process extract shown in Figure 8.1, where the flow of an $Application$ is represented with routed data flow, while a $Job\ Opening$ is read from a repository depicted with a cylinder. Specification of object states in a process model is a recently-added feature of WBM and is currently directly supported only for routed data flow. As shown in Figure 8.3(a), states are displayed on typed edges in a process model. WBM currently restricts the user to specifying one state per edge. As a result, multiple produced states of an action cannot be modeled directly, but can be emulated by placing a decision to follow the action in the data flow and assigning different states to the outgoing edges of the decision. This is shown in Figure 8.3(a), where the interpretation is that the $Register\ and\ Evaluate$ action has two produced states for $Claim$, namely $Granted$ and $Rejected$.

[3] `http://www.alphaworks.ibm.com`

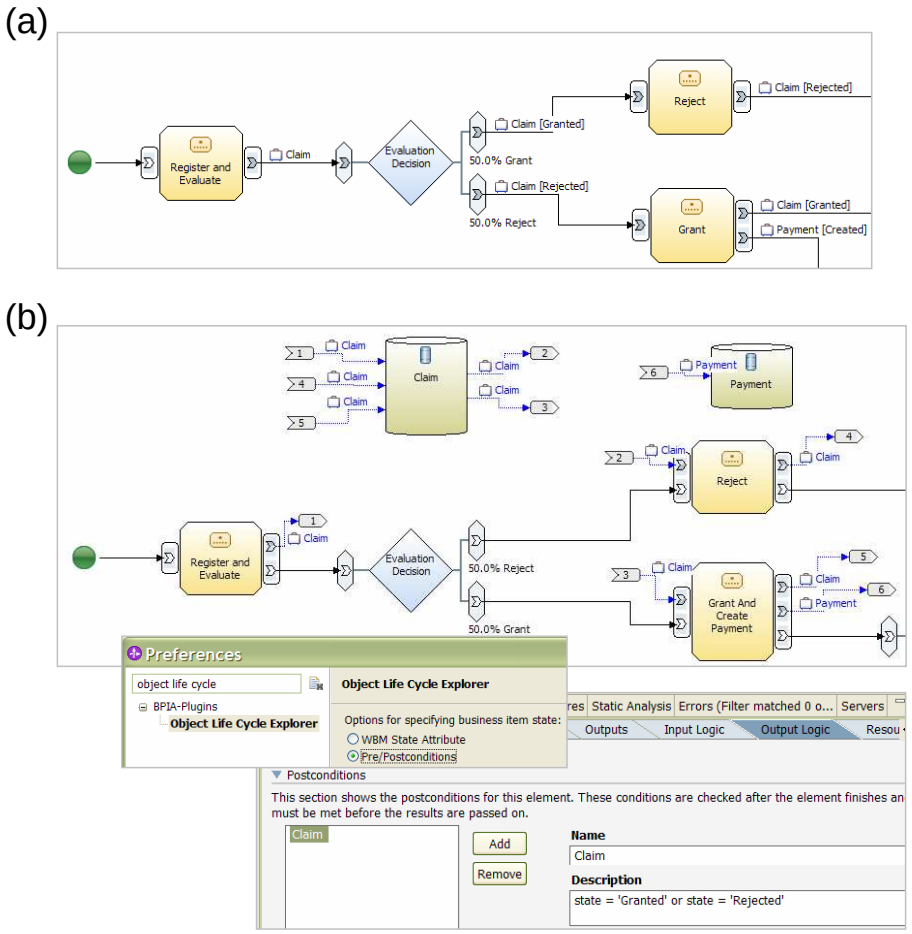


Figure 8.3: Data flow and state specification in WBM process models

An example only using repository data flow is shown in Figure 8.3(b). In such models, states cannot be assigned to edges. Instead, we use the pre- and post-conditions of actions to capture their accepted and produced states, respectively. As shown in Figure 8.3(b), a preference option allows the user to indicate whether the native WBM State Attribute or the Pre/Postcondition approach is used to specify object states. Process models with mixed data flow and state specification approaches can be handled, but states not specified according to the approach indicated in the preferences are ignored.

Specification of dependency state sets is not directly supported in WBM and therefore our implementation does not take these into account.

8.1.2 Consistency Checking

For checking the correctness of state specifications in process models, Object Life Cycle Explorer implements the algorithm for the computation of effective input and output states described in Chapter 4. This check is initiated every time the user invokes the check for inter-model consistency or the extraction of object life cycle models from process models, since these features also make use of the computed effective states. Problems detected during the check are logged as warnings.

The inter-model consistency check can be invoked on several process and object life cycle models at the same time, as can be seen in the wizard shown in Figure 8.4.

Consistency conditions defined in Chapter 4 are evaluated for each pair of process and object life cycle models in the selection, producing an aggregated set of inconsistencies as a result. Several *subprocess traversal* options are offered to the user: no traversal (Do not traverse subprocesses) treats subprocesses as atomic actions without expanding their contents; local traversal (Traverse local processes only) expands the contents of local subprocesses that are defined within the selected process models; and full traversal (Traverse local and global subprocesses) expands the contents of subprocesses defined locally within the process models and globally within the project. Different subprocess traversal options can lead to different induced transitions, first and last states being computed for objects and can therefore result in different inconsistencies.

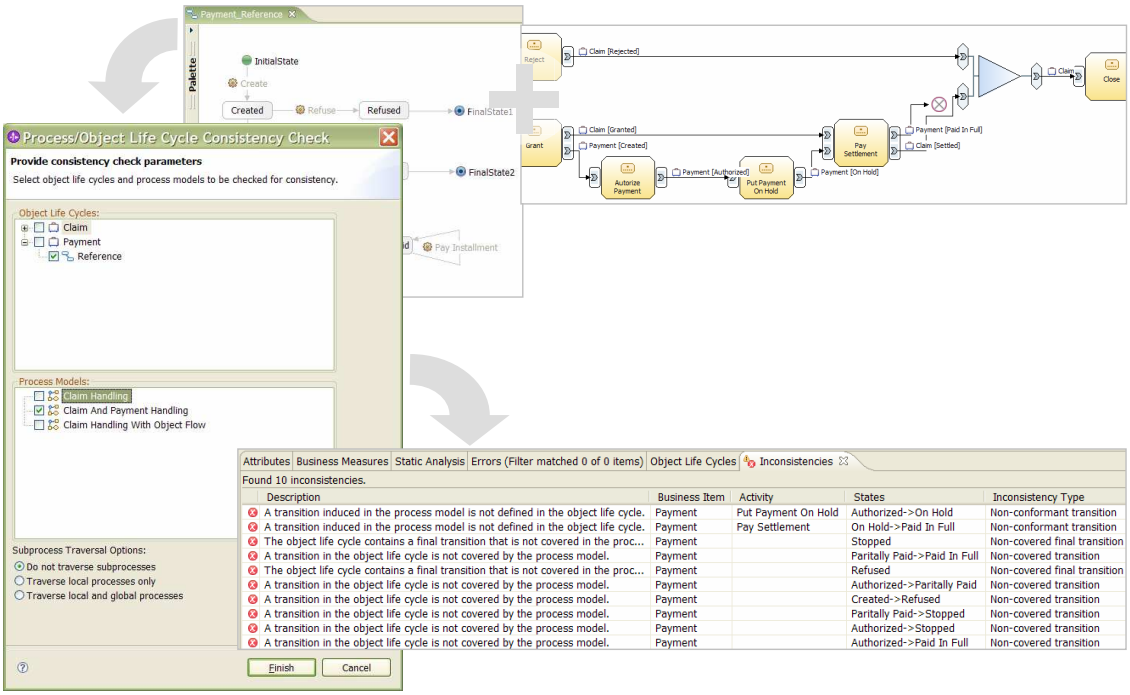


Figure 8.4: Consistency checking in Object Life Cycle Explorer

8.1.3 Inconsistency Resolution

Using our approach to the design and implementation of inconsistency resolutions with side-effect expressions, a sample set of 8 resolutions were implemented as part of Object Life Cycle Explorer by a student during an internship [Monot, 2008]. The goal here was *not* to provide a complete set of possible resolutions, but rather to demonstrate the feasibility of developing inconsistency resolutions using our approach and facilitate validation of the potential benefits of the approach to the modeler.

All the implemented resolutions update a process model in the context of a particular inconsistency. A brief description of each resolution is given below, while the interested reader is referred to Object Life Cycle Explorer user guide [Wahler et al., 2008] for more details.

- R1/R2: Removes data from incoming/outgoing connections of a process node.
- R3/R4: Removes state from incoming/outgoing connections of a process node.

- R5/R6: Associates data and state with incoming/outgoing connections of a process node.
- R7/R8: Inserts a new task between a process node and its predecessor/successor nodes.

Table 8.2: Reuse of resolutions across inconsistency types

Inconsistency type / Resolution	R1	R2	R3	R4	R5	R6	R7	R8
non-conformant transition	+	+	+	+			+	+
non-conformant initial transition		+		+	+		+	+
non-conformant final transition	+		+			+	+	+
non-covered transition					+	+	+	+
non-covered initial transition	+					+	+	+
non-covered final transition		+			+		+	+

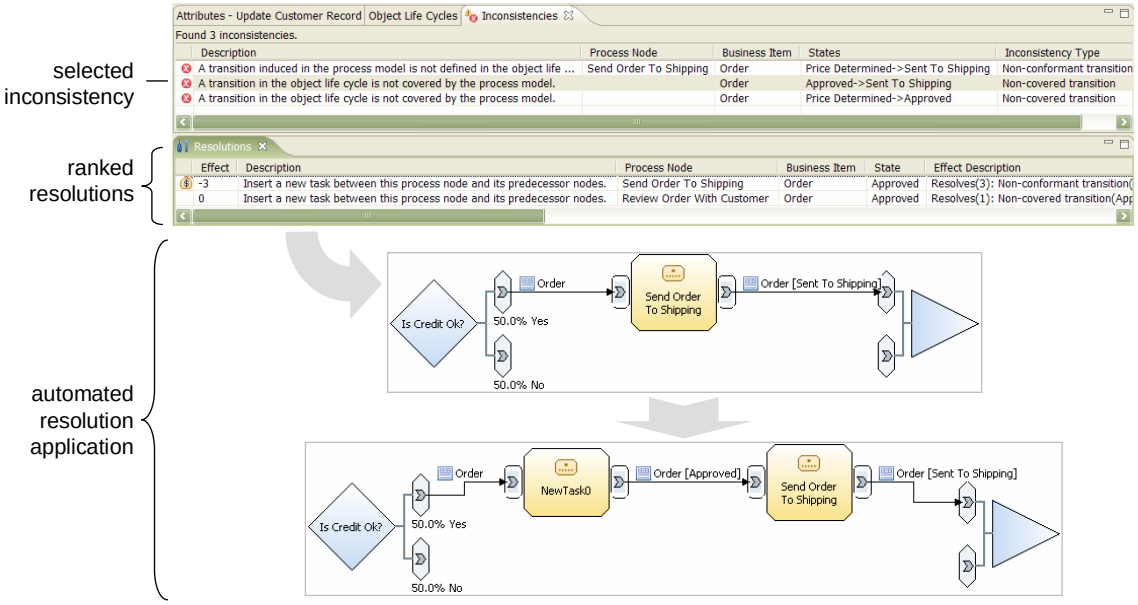


Figure 8.5: Inconsistency resolution in Object Life Cycle Explorer

Most resolutions can be applied to resolve inconsistencies of different types, as shown in Table 8.2. This generally results in several alternative resolutions being available to resolve a particular inconsistency. After running the consistency check on a set of process and object life cycle models, the user can view available resolutions by selecting an inconsistency in the inconsistencies view, as shown in Figure 8.5. In accordance with our solution, the alternative resolutions are ranked according to their side-effects. Resolutions that remove more inconsistencies then they introduce, i.e. those with a negative overall effect, are marked with a money bag icon. The details of the resolution side-effects are also shown to the user in the resolutions view, so that he/she can examine these before applying any actual changes to the models. Once decided among the alternatives, the user can invoke the chosen resolution, which automatically updates the corresponding process model. An example of such an update is illustrated in Figure 8.5.

8.1.4 Model Transformations

Object life cycle extraction can be invoked on a selected process model via the object life cycle menu, as shown in Figure 8.6. The object life cycle extraction transformation described in Chapter 6 is implemented to generate an object life cycle model for each object type (business item in WBM) with a non-empty state specification in the selected process model. In the example shown in Figure 8.6, two object life cycle models for *Customer Record* and *Order* are extracted from the *Customer Order Handling* process model. A list of the extracted object life cycle models is displayed in the object life cycles view, from which the user can open an object life cycle model either in the object life cycle editor or in the state transition table view. Figure 8.6 shows that the state transition table view additionally displays the type of action that gave rise to each state transition (denoted by different icons in the Activity column) and the process model used to generate the object life cycle model (In Process column). The object life cycle extraction wizard (not shown here) allows the user to select a subprocess traversal option, similar to the option offered for consistency checking. This allows the modeler to generate object life cycle models at different levels of granularity.

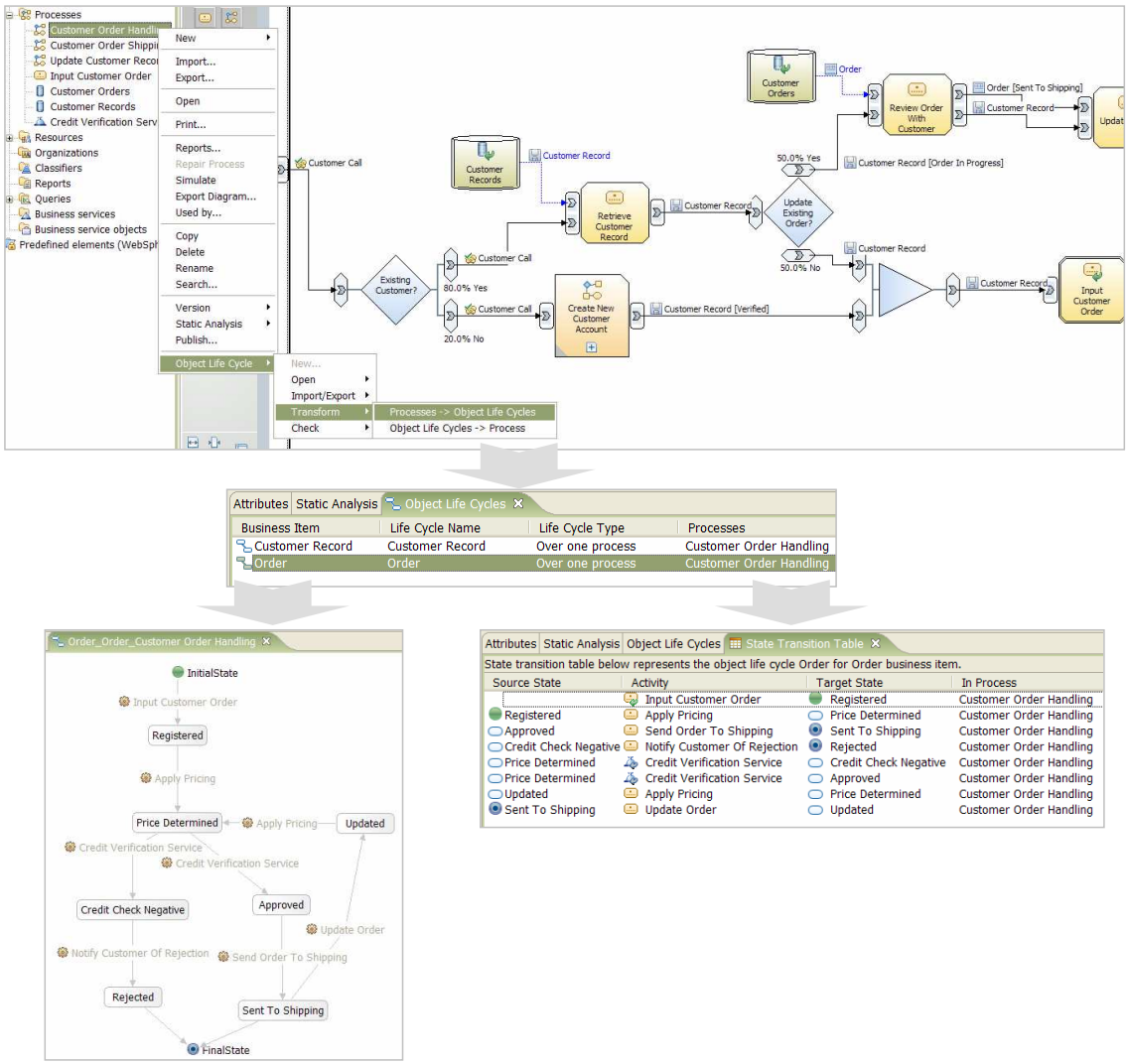


Figure 8.6: Object life cycle extraction in Object Life Cycle Explorer

The process model generation wizard allows the modeler to select one or more object life cycle models to be used in the generation of a new process model, as shown in Figure 8.7. Synchronization events must be introduced manually into the selected object life cycle models prior to the generation. The generation produces process models with repository data flow, where state specifications are captured using pre- and post-conditions as described in Section 9.3.1. The transformation from repository to routed data flow in the generated process models is not implemented in Object Life Cycle Explorer. Instead, the actual repositories are omitted in the generated process model to make it simpler for the modeler to manually turn the data flow into the repository or routed form during the customization.

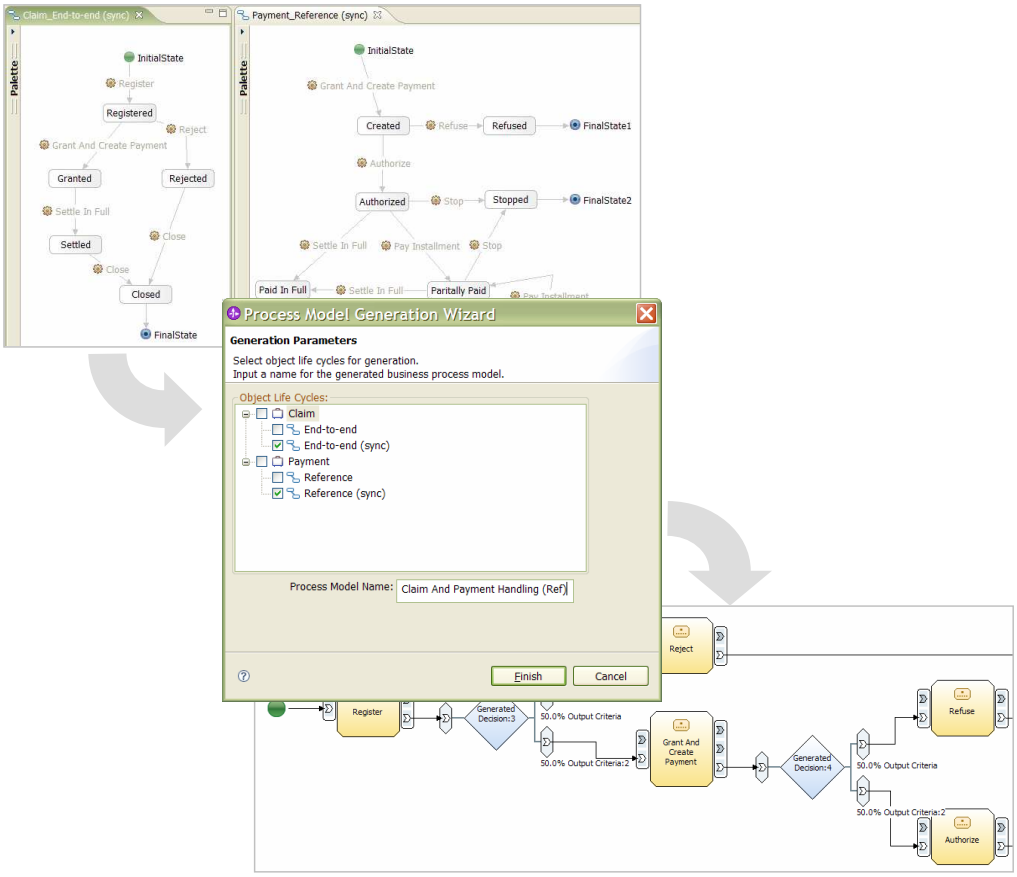


Figure 8.7: Process model generation in Object Life Cycle Explorer

8.1.5 Other Features

Although we did not implement the computation of the expected coupling in the context of transiting to object-centric process implementations, the foundation for this transition is already there owing to the integration of Object Life Cycle Explorer with IBM WebSphere Integration Developer (WID)[4]. WID offers extensive support for the implementation of business process logic using a service-oriented approach. Using the native functionality of WBM, it is possible to export process models created in WBM to BPEL [BPEL, 2003] for further development in WID (see Figure 8.8). This supports the so-called activity-centric

[4] `http://www.ibm.com/software/integration/wid`

process implementation.

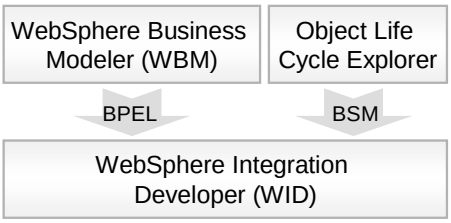


Figure 8.8: Integration with WID

As already described in Chapter 7, WID also offers Business State Machines (BSMs) [Beers and Carey, 2006] alongside BPEL, which can be used for object-centric process implementation. Since Object Life Cycle Explorer stores object life cycle models according to the same meta-model used for BSMs in WID, object life cycle models can be easily transferred to BSMs in WID, as illustrated in Figure 8.8. These BSMs can serve as skeletons of deployable object life cycle components, which need to be further refined by resolving the operations associated with transitions to service interface operations, etc. In a full-blown implementation of our solution, this transition can be further enhanced with the computation of the expected coupling and process model revisions as described in Chapter 7.

8.2 Modeling Strategies

Our solution for integrated process and object life cycle modeling comprises several aspects, including consistency checking, inconsistency resolution and model transformations. Different scenarios may place more emphasis on some of these aspects than on others. For example, consider a scenario where process modeling and object life cycle modeling are assigned to different people on a project who work independently from each other. In this case, consistency checking and inconsistency resolution are essential to coordinate the work done by these two people. However, model transformations are of a lesser importance. On the other hand, the object life cycle extraction can be instrumental in another scenario where one person is responsible for modeling the required process logic and object state evolution. Instead of modeling both, processes and object life cycles, only process models be created and subsequently used to automatically extract the object life cycle models.

Instead of having one universal method that tries to address all possible scenarios, we propose several *modeling strategies*, each of which is targeted at one specific scenario. The modeling strategies can be combined in a project that spans requirements of several different scenarios. We can imagine that new strategies can also be later developed to capture best practices of integrated process and object life cycle modeling.

We define three modeling strategies, namely *Modeling-In-Parallel*, *Validate-Check-Resolve* and *Reference-Driven Process Modeling*. Each of these is described next.

8.2.1 Modeling-In-Parallel

The scenario addressed by the Modeling-In-Parallel strategy concerns situations where process and object life cycle modeling is performed by different roles, which may or may not be fulfilled by the same person. To ensure that the independently-developed models

that represent different views on the application are consistent, regular *coordination points* should be put into practice in this scenario. As illustrated in Figure 8.9, each coordination point involves checking consistency of the process and object life cycle models and resolving the detected inconsistencies (steps 2 and 3). It is possible that both, process and object life cycle models, are changed to establish consistency during the inconsistency resolution. Decisions required during the inconsistency resolution should be agreed upon by both of the roles involved. Coordination of models developed by different persons can be realized in Object Life Cycle Explorer by exporting, exchanging and importing the necessary models. However, consistency checking and inconsistency resolution support should ideally be integrated with a version control system to fully support distributed modeling in this scenario.

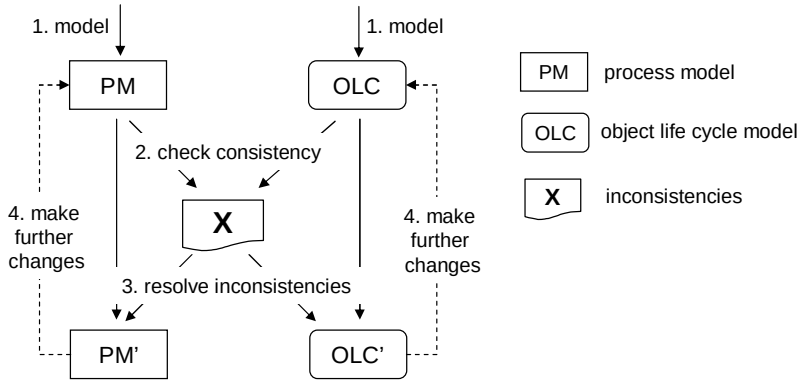


Figure 8.9: Modeling-In-Parallel strategy

8.2.2 Validate-Check-Resolve

The Validate-Check-Resolve strategy addresses a scenario where process models need to be developed to not only capture particular process logic, but also the required object state evolution. In this scenario, the creation of object life cycle models is optional, however they are used as a tool for validating the state evolution implicitly captured in the created process models. As illustrated in Figure 8.10, object life cycle models are first extracted from the process models and then validated by the modeler (steps 2 and 3). For the manual validation process, we propose a systematic approach we call the *5-Step Object Life Cycle Validation* method. The method describes how the modeler should examine elements of an extracted object life cycle model, comparing them to his/her intention for the state evolution of the underlying object type.

The steps of the 5-Step Object Life Cycle Validation method (refinement of step 3 in Figure 8.10) are as follows:

3.1 Examine each initial transition for validity. An initial transition is valid if objects of this object type should be either created in the target state of this transition inside the process model or they should be passed to the process model in this state via an input parameter.

3.2 Examine each final transition for validity. A final transition is valid if the source state of this transition should indeed be the last state that objects of this object type reach in this process model.

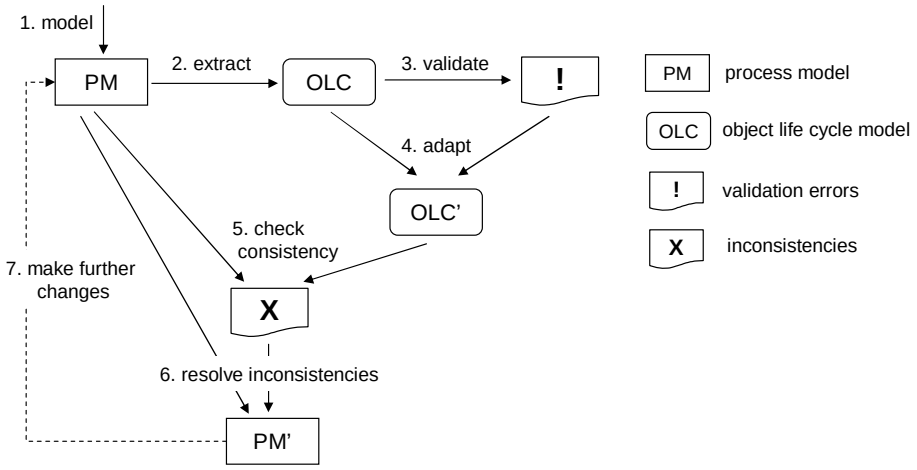


Figure 8.10: Validate-Check-Resolve strategy

3.3 Examine each intermediate transition for validity. Check that the action associated with each intermediate transition should indeed change the source state of objects of this object type to the target state.

3.4 Check transition splits for validity. A transition split means that more than one action can change the state of objects of this object type from the same source state. Ensure that each such choice is indeed intended.

3.5 Find missing transitions. Determine whether actions in the process model should induce other transitions on objects of this object type. Identify any missing actions in the process model that should induce transitions on objects of this object type.

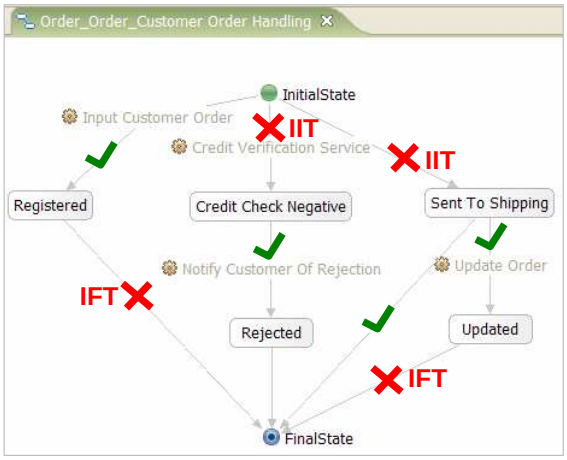


Figure 8.11: Example of validation errors in an object life cycle model

Each validation step can give rise to several validation errors of the following types: *invalid initial transition (IIT)*, *invalid final transition (IFT)*, *invalid transition (IT)*, *invalid transition split (ITS)* and *missing transition (MT)*. These validation errors should be recorded for use in the subsequent steps in the modeling strategy. Figure 8.11 shows an example object life cycle model for object type $Order$ extracted from the $Customer\ Order\ Handling$ process model, and four validation errors discovered during the 5-Step

Object Life Cycle Validation method. More details and examples for the 5-Step Object Life Cycle Validation method can be found in the user guide of Object Life Cycle Explorer [Wahler et al., 2008].

In step 4 of the Validate-Check-Resolve strategy illustrated in Figure 8.10, the extracted object life cycle models are manually adjusted to fix the validation errors. For example, fixing an invalid initial transition involves removing the corresponding transition and possibly replacing it with another non-initial transition in the object life cycle model. Such adjustments clearly introduce inconsistencies between the original process models and the adapted object life cycle models. The changes are then propagated to the process models in a semi-automated manner, by first running the consistency check in step 5 and then resolving the detected inconsistencies in step 6. This approach of validation followed by consistency checking and inconsistency resolution gives rise to the name of this modeling strategy: Validate-Check-Resolve.

8.2.3 Reference-Driven Process Modeling

The Reference-Driven Process Modeling strategy addresses a scenario where a process model that is consistent with some already existing object life cycle models is to be obtained. The existing object life cycle models serve as a reference for the process model. Such reference object life cycle models may embody an industry standard or a best practice; may be harvested from previous projects or developed to capture the requirements of the current project.

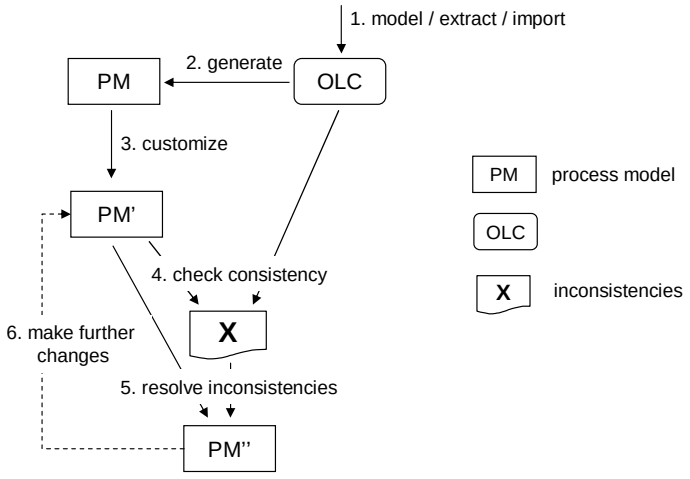


Figure 8.12: Reference-Driven Process Modeling strategy

According to this modeling strategy, the process model generation is applied first to produce an initial process model that is consistent with the given object life cycle models (step 2 in Figure 8.12). The generated process model can be customized by adding further details to it (step 3), after which it is necessary to check the consistency with regards to the original object life cycle models and resolve the inconsistencies introduced during the customization (steps 4 and 5).

8.3 Summary and Discussion

In this chapter, we presented the tool and method developed to support our framework for integrated process and object life cycle modeling. We showed that almost all aspects of the proposed solution were implemented in our prototype and subsequently released for public use on IBM alphaWorks. The modeling strategies presented here were used as a basis for the documentation and tutorials included in the alphaWorks download package. The initial feedback on the prototype and its documentation received from various modelers was very positive. It demonstrated a significant interest in our solution and acceptance of the way it was implemented in the tool and presented in the supporting method.

Now that we have shown the feasibility of our framework, we proceed to describing the validation of our solution with respect to other criteria such as effectiveness and added value.

Chapter 9

Validation

In this chapter, we present two case studies carried out to validate our framework for integrated process and object life cycle modeling. We begin by motivating our choice of case studies as the validation method, presenting the validation hypotheses and describing the case study setting in Section 9.1. We then describe the first case study in Section 9.2, the subject of which was an ongoing project aimed at streamlining and automation of management processes in the context of Computer-Aided Design. In this case study, we demonstrate that our solution facilitates the creation of new models that correctly capture the process logic and state evolution of objects intended by the modeler. The second case study, described in Section 9.3, was focused on a collection of IBM reference models for the insurance industry. In this case study, we show that our solution is instrumental in obtaining consistency of existing process and object life cycle models. After evaluating the results of the two case studies, we discuss the threats to the validity of these results in Section 9.4.

9.1 Validation Approach

The most common methods for validation of approaches in the area of software engineering are experiments, surveys and case studies [Kitchenham et al., 1995]. Each of these methods is more or less suitable for different validation scenarios and yields different types of results. Experiments [Wohlin et al., 1999] are characterized by a high degree of control over many variables in a study, leading to a high degree of confidence in the experiment results. The difficulty of implementing the necessary controls on a large scale usually leads to experiments being relatively small in size and short in duration. Surveys involve the collection of data from multiple projects and are useful for testing the generalization of claims. However, applying surveys for the evaluation of new approaches is usually challenging, since a wide adoption of the evaluated approach is required in a number of comparable projects. Finally, case studies [Yin, 2002] are in-depth studies of one or several selected projects that usually represent a typical industrial scenario for the application of the approach in question. Although it is usually difficult to generalize results of a case study, this validation method is useful for disproving hypotheses and getting a broad range of insights about applying an approach in a realistic setting. Validation can also be performed by combining some of these methods.

Our validation goal is to determine whether our framework for integrated process and object life cycle modeling does indeed improve the state-of-the art. As most of the aspects

of our solution have been implemented in Object Life Cycle Explorer, we focus on evaluating this tool and its supporting method as the embodiment of our solution (from now on "solution" refers to this tool and method). More specifically, we consider the released version of Object Life Cycle Explorer (see Section 8.1 in Chapter 8).

Since Object Life Cycle Explorer implements several functional requirements and the method suggests several modeling strategies, it would be difficult to design an experiment evaluating all these aspects. A survey of different projects applying our solution is also challenging, since a sufficiently wide adoption of our approach needs a longer period of time. On the other hand, case studies present us with an opportunity of studying the use of our solution in an industrial project and obtaining results that may be not completely generalizable, but are of a high credibility and relevance for similar project settings. For these reasons, we choose case studies to validate our solution.

9.1.1 Validation Hypotheses

Our general claim is that our solution facilitates the creation of process and object life cycle models that are *valid* and *consistent*, which cannot be easily achieved using state-of-the-art approaches. In this context, model validity describes the extent to which a given model captures the intent of the modeler, while model consistency means that a set of given models satisfy inter-model consistency as described in Chapter 4. Additionally, we claim that our solution alleviates the challenges facing the modeler during the resolution of model inconsistencies. Here, we aim to evaluate the comparison of alternative resolutions based on side-effect forecast, which is the part of our inconsistency resolution approach released in Object Life Cycle Explorer.

We formulate the following hypotheses to be tested in the case studies:

- H0: A state-of-the-art approach to process and object life cycle modeling is sufficient to produce *valid* and *consistent* models.
- H1: Our solution *improves the validity* of process and object life cycle models developed using a state-of-the-art approach.
- H2: Our solution *improves the consistency* of process and object life cycle models developed using a state-of-the-art approach.
- H3: Our solution *reduces the modeler effort* required during the resolution of model inconsistencies.

H0 is the *null hypothesis* for the validation, the rejection of which is necessary to consider the truth of hypotheses H1 and H2, which together comprise the *alternative hypothesis*. H0, H1 and H2 are concerned with the integration of process and object life cycle modeling. Additionally, we formulate hypothesis H3 to evaluate our solution to inconsistency resolution, which can also be applied to other types of models. Using case studies, we cannot formally prove H1, H2 or H3, but we can show they hold in the particular setting of our case studies and further argue about their general applicability.

9.1.2 Case Study Setting

In this chapter, we describe two case studies that we carried out to validate our solution. The first case study assesses the value of our solution during the development of new

models that are required to capture specific process logic and object state evolution. Using the Validate-Check-Resolve modeling strategy, we show that a set of models created by an experienced modeler using a state-of-the-art approach do not capture the intended state evolution of objects correctly. Furthermore, we demonstrate how the consistency check and inconsistency resolution in Object Life Cycle Explorer enable the alignment of these models with the object state evolution requirements. The second case study evaluates the extent to which our solution can improve the consistency of a set of existing process and object life cycle models created using state-of-the-art modeling tools. In this case, we demonstrate that Object Life Cycle Explorer facilitates the location and classification of inconsistencies in the given models, as well as assists in the resolution of these inconsistencies.

In the first case study, our solution is applied in an ongoing project concerned with the streamlining and automation of Computer-Aided Design (CAD) management processes. The second case study is performed on a set of IBM industry reference models for the insurance industry, namely the IBM Insurance Application Architecture (IAA)[1]. The details of the case studies and the obtained results are described in the following sections.

9.2 Case Study 1: CAD Management

This case study is concerned with a proof of concept phase of a project for a manufacturer of large and complex machinery (the name of the company is omitted for confidentiality purposes). The project focuses on the design sharing and change management processes in the context of CAD. Streamlining and automation of these processes is the final goal of the project, while the proof of concept phase is mostly concerned with developing the as-is and to-be process models.

Each process area, design sharing and change management, identifies one main object that is manipulated throughout these processes. States of these objects represent important processing milestones and therefore, understanding the complete state evolution of the objects throughout the processes is important for the client. Furthermore, the functionality of different process parts is understood in terms of the object state changes they incur. Therefore, the models developed in this project are required to correctly capture not only the logic of the processes, but also the state evolution of the main objects.

Our goal in this case study is to assess the value of our solution during the development of new models. Our general claim is that our solution facilitates the development of models that are *valid* with respect to the process logic and object state evolution intended by the modeler. We focus on validation hypotheses H0, H1 and H3 defined in Section 9.1.1.

We consider the IBM WebSphere Business Modeler (WBM) to embody a state-of-the-art approach that allows one to model process logic and object state evolution using process models with state specifications. The case study was performed in a collaboration with the process architect responsible for the modeling on the project, who was already an experienced user of WBM and was additionally equipped with Object Life Cycle Explorer and its supporting method. To diminish the lack of experience as a confounding factor of the case study, the process architect performed several tutorials to learn how to use our solution prior to the project.

The Validate-Check-Resolve modeling strategy was selected by the process architect as the method basis for the organization of modeling and validation activities during

[1] `http://www.ibm.com/industries/financialservices/doc/content/solution/278918103.html`

the project. The modeling was performed by the process architect himself, while the validation and subsequent revision steps were performed with our support. Throughout these activities, we recorded quantitative data to capture the direct effects of using our solution on the resulting models.

In the following, we first provide an overview of the models that were subject to the study, explain the applied method in more detail, present the recorded data, and finally evaluate our results.

9.2.1 Models and Method

Table 9.1 gives statistics for the process models developed to represent the to-be process of sharing so-called design workpackages. The models are divided into two use cases: use case 1.1 is the simple design workpackage exchange process, while use case 1.2 is an extension of use case 1.1 with change control. As shown in the table, use case 1.1 comprises 4 process models that form a composition hierarchy with 2 levels. These process models contain a total of 41 actions, including atomic actions and those actions that reference other processes. The only object type used in the process models is $Workpackage$, which has 7 states defined, excluding initial and final states. Use case 1.2 comprises 13 new actions and reuses 38 actions from use case 1.1 by referencing its processes, leading to a total of 51 actions.

Table 9.1: Model statistics for CAD Management

Use case	Process models	Actions	Deepest hierarchy	Object types	States per type
1.1	4	41	2	1	7
1.2	6	51	4	1	7

All process models use routed data flow, as shown in a process extract in Figure 9.1. The process models contained decisions and merges, but no explicit forks and joins. Instead, control and data flows were forked and joined implicitly via multiple outgoing and incoming edges of actions, also illustrated in the process extract in Figure 9.1.

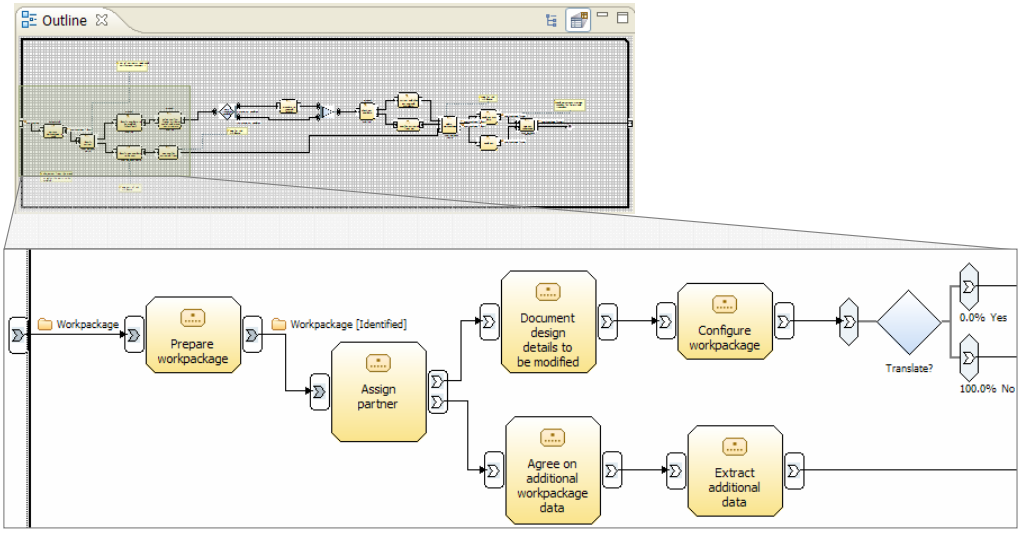


Figure 9.1: Example process overview and extract for CAD Management

An overview of how the Validate-Check-Resolve modeling strategy was applied in the case study is given in Figure 9.2. The process modeling (step 1), including the definition of object states and the specification of object states in process models, was performed solely by the process architect using WBM. Object life cycle extraction and validation (steps 2 and 3) were performed with our assistance in an interactive session with the process architect. The 5-Step OLC Validation method (see Section 8.2.2) was applied to systematically identify validation errors in the extracted object life cycle models. The source of each error and an appropriate correction of the process models were discussed with the process architect before moving onto steps 4, 5 and 6. In this way, we established the required changes of the process models in a tool-independent manner, such that we could assess the extent to which Object Life Cycle Explorer can automatically locate and apply these changes. The desired object life cycle model was agreed upon with the process architect (step 4), after which steps 5 and 6 were performed by us based on the knowledge about how the process models should be changed. At the end, the corrected process models were given back to the process architect to be refined and extended in further modeling iterations.

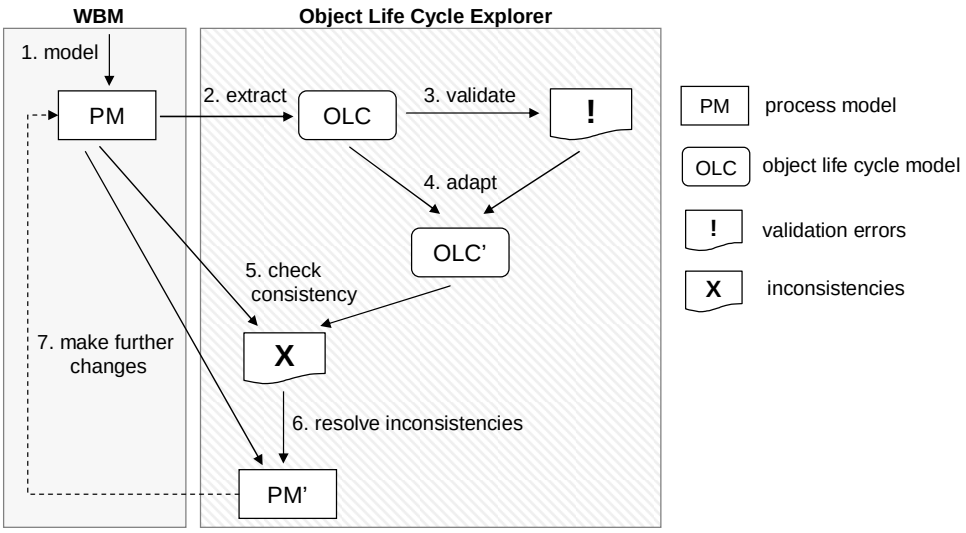
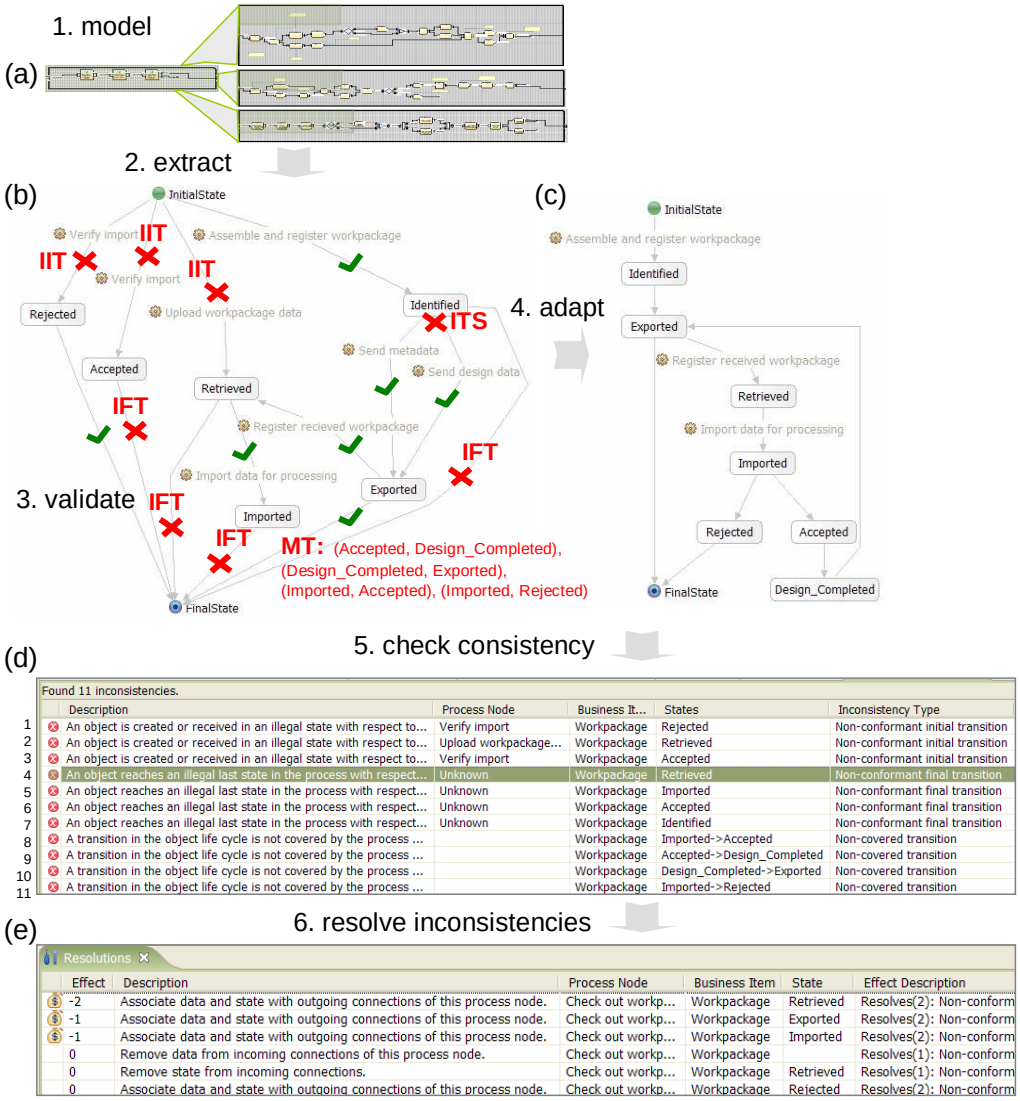


Figure 9.2: Validate-Check-Resolve modeling strategy applied in case study 1

In the following, we demonstrate the steps of the method with some examples from the case study shown in Figure 9.3. The process models for use case 1.1 shown in Figure 9.3(a) were used to extract an object life cycle model for the $Workpackage$ object type shown in Figure 9.3(b). The 5-Step OLC Validation method applied to this object life cycle model yielded validation errors of different types, including an invalid initial transition (IIT) to state $Rejected$, an invalid transition split (ITS) from $Identified$ to $Exported$, and missing transitions (MT) from $Accepted$ to $Design\_Completed$ and from $Design\_Completed$ to $Exported$. After validation, the extracted object life cycle model was adapted to capture the desired state evolution of the $Workpackage$ shown in Figure 9.3(c).

Running the consistency check with a complete subprocess traversal on the parent process model and the adapted object life cycle model located 11 inconsistencies shown in Figure 9.3(d). Remember that in the Validate-Check-Resolve modeling strategy, inconsistencies are “desirable” in the sense that they facilitate the propagation of the object life cycle model changes to the process model. The relationship between the validation errors



	Description	Process Node	Business It...	States	Inconsistency Type
1	An object is created or received in an illegal state with respect to...	Verify import	Workpackage	Rejected	Non-conformant initial transition
2	An object is created or received in an illegal state with respect to...	Upload workpackage...	Workpackage	Retrieved	Non-conformant initial transition
3	An object is created or received in an illegal state with respect to...	Verify import	Workpackage	Accepted	Non-conformant initial transition
4	An object reaches an illegal last state in the process with respect...	Unknown	Workpackage	Retrieved	Non-conformant final transition
5	An object reaches an illegal last state in the process with respect...	Unknown	Workpackage	Imported	Non-conformant final transition
6	An object reaches an illegal last state in the process with respect...	Unknown	Workpackage	Accepted	Non-conformant final transition
7	An object reaches an illegal last state in the process with respect...	Unknown	Workpackage	Identified	Non-conformant final transition
8	A transition in the object life cycle is not covered by the process ...		Workpackage	Imported->Accepted	Non-covered transition
9	A transition in the object life cycle is not covered by the process ...		Workpackage	Accepted->Design_Completed	Non-covered transition
10	A transition in the object life cycle is not covered by the process ...		Workpackage	Design_Completed->Exported	Non-covered transition
11	A transition in the object life cycle is not covered by the process ...		Workpackage	Imported->Rejected	Non-covered transition

	Effect	Description	Process Node	Business Item	State	Effect Description
	-2	Associate data and state with outgoing connections of this process node.	Check out workp...	Workpackage	Retrieved	Resolves(2): Non-conform
	-1	Associate data and state with outgoing connections of this process node.	Check out workp...	Workpackage	Exported	Resolves(2): Non-conform
	-1	Associate data and state with outgoing connections of this process node.	Check out workp...	Workpackage	Imported	Resolves(2): Non-conform
	0	Remove data from incoming connections of this process node.	Check out workp...	Workpackage		Resolves(1): Non-conform
	0	Remove state from incoming connections.	Check out workp...	Workpackage	Retrieved	Resolves(1): Non-conform
	0	Associate data and state with outgoing connections of this process node.	Check out workp...	Workpackage	Rejected	Resolves(2): Non-conform

Figure 9.3: Method illustrated with examples from CAD Management

and the inconsistencies arising from the changes made to fix the object life cycle model is evident: resolving invalid initial/final transitions in the object life cycle model introduces non-conformant initial/final transition inconsistencies, while adding missing transitions to the object life cycle model introduces non-covered transition inconsistencies. However, there is one validation error, fixing which in the object life cycle model did not introduce any inconsistencies. This is the invalid transition split from state *Identified* to *Exported*. Removing one of these transitions in the object life cycle model does not affect the object life cycle conformance or coverage of the original process models. We demonstrate how this error is handled later in this section.

Figure 9.3(e) shows the available resolutions for the selected inconsistency numbered 4 in Figure 9.3(d), i.e. the non-conformant final transition to state *Retrieved*. In total, 15 resolutions are available and these are shown to the modeler ranked according to their side-effects. We decided not to use cost-reduction in this case study, since all inconsistency types were deemed equally important.

Figure 9.4(a) shows the context of the first inconsistency resolution from Figure 9.3(e)

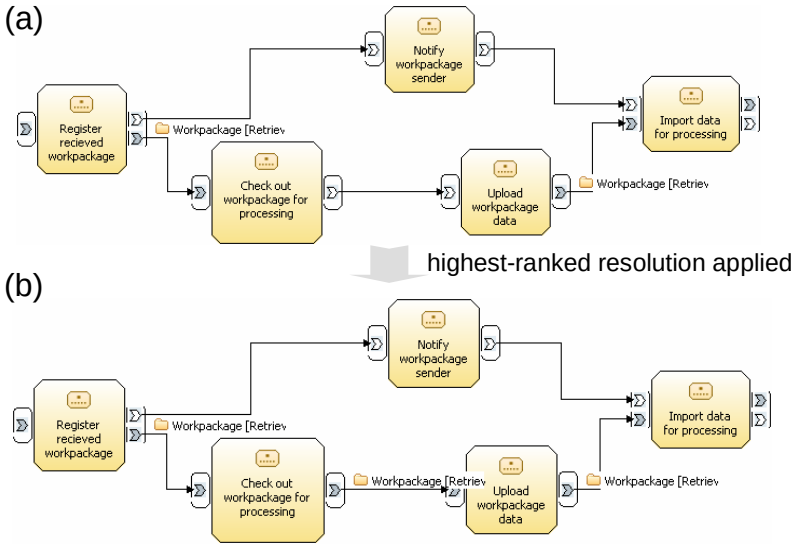


Figure 9.4: Example resolution

in the process model: the *Check out workpackage for processing* action receives a copy of the *Workpackage* in state *Retrieved* and does not pass it on further in the process; thus *Retrieved* is interpreted as a last state of *Workpackage*. During the validation step, we discussed the reasons for the validation errors with the process architect and established that in this particular case, the *Workpackage* should be passed from the *Check out workpackage for processing* action to the *Upload workpackage data* action. As can be seen in Figure 9.3(e), this exact change is offered as the highest-ranked resolution by Object Life Cycle Explorer. The resolution does not only resolve this inconsistency, but also inconsistency numbered 2 in Figure 9.3(d), i.e. the non-conformant initial transition to state *Retrieved*. Since the resolution does not induce any new inconsistencies, its total effect value is -2. Applying this resolution automatically, we obtained the model shown in Figure 9.4(b).

In the case study, we also encountered an example of an inconsistency that could not be resolved using the resolutions available in Object Life Cycle Explorer in a straightforward manner. This example is inconsistency numbered 7 in Figure 9.3(d), i.e. the non-conformant final transition to state *Identified*. The context of this inconsistency in the process model is shown in Figure 9.5(a) (see *Assign partner* action). Together with the process architect, we established that the *Workpackage* should be passed from the *Assign partner* action, through several intermediate nodes, all the way to the *Assemble and register workpackage* action. Since the resolutions currently implemented in Object Life Cycle Explorer apply only local changes such as adding data and state to one edge, this inconsistency could only be resolved by applying a series of resolutions. In this particular example, it was easier to simply adapt the process model manually. It would however be possible to automate such a compound resolution and to specify its side-effect expression, thus extending Object Life Cycle Explorer.

As already mentioned, the invalid transition split from state *Identified* to *Exported* could not be fixed through the consistency check and inconsistency resolution. It was fixed manually, as shown in Figure 9.6. In the original process model, the state of the *Workpackage* was updated to *Exported* by two parallel actions, *Send metadata* and *Send design data*. However, the *Workpackage* should only be considered *Exported* once both of these actions have completed. We removed the state *Exported* from the pro-

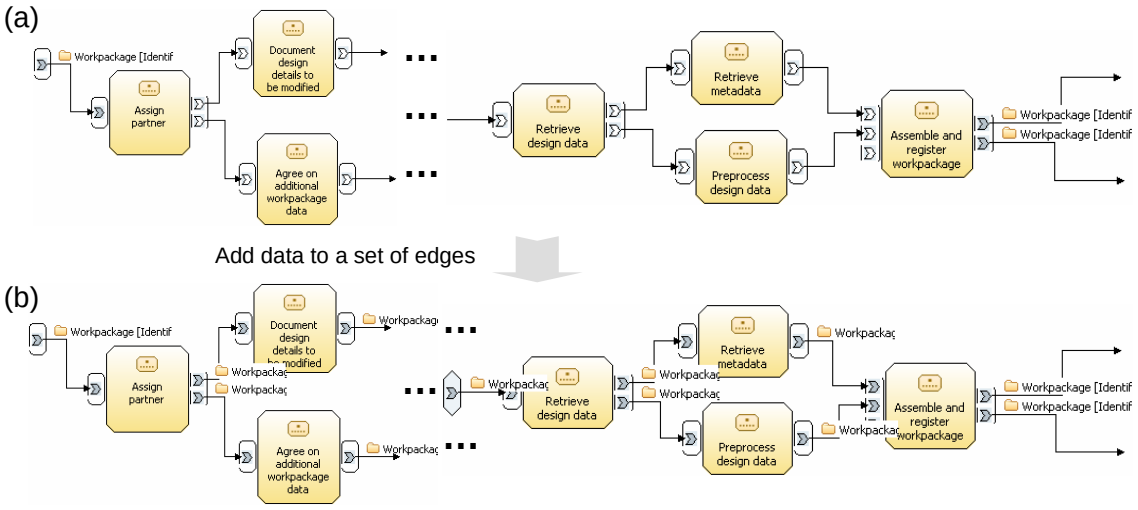


Figure 9.5: Example of a complex resolution

duced states of these two actions, as shown in Figure 9.6, which resulted in the state of *Workpackage* only being updated to *Exported* by the *Register workpackage submission* action, as desired.

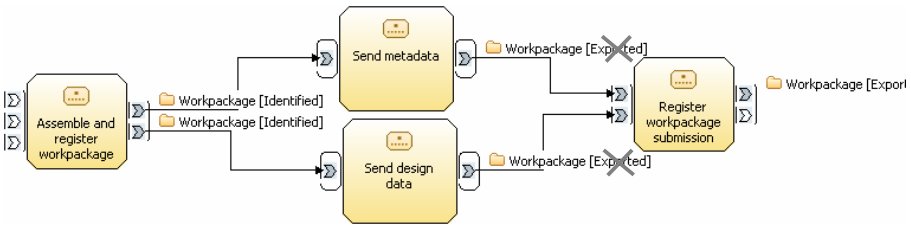


Figure 9.6: Resolving an invalid transition split

In the next section, we present the quantitative data recorded for the whole case study.

9.2.2 Quantitative Data

The aggregated quantitative data for the case study is presented in Table 9.2. Apart from the number of discovered validation errors, the total number of subsequently introduced inconsistencies is shown, which also includes those inconsistencies that were induced as a side-effect during the inconsistency resolution process. The table also gives the percentages of those inconsistencies that were resolved completely automatically by applying resolutions in Object Life Cycle Explorer. The percentage of the manual resolutions that can be implemented as automated resolutions in a straightforward manner is also provided. The judgement of automation is of course subjective, but still provides us with an insight into the nature of manual resolutions applied. Furthermore, the percentages of total inconsistencies that were resolved/induced as a side-effect of applying a resolution to another inconsistency are given.

The *resolution choice reduction* measures the extent to which the resolution ranking based on side-effects assisted the modeler in choosing the appropriate resolution. For each inconsistency in the case study, we looked through the ranked resolution list for the resolution that entailed making changes to the process model that were agreed upon

Table 9.2: Aggregated results for CAD Management

	Use case 1.1	Use case 1.2	Weighted average
Validation errors	12	1	-
Inconsistencies	12	1	-
Automatically resolved inconsistencies	92%	0%	84%
Automatable manual resolutions	100%	0%	50%
Resolved inconsistencies as side-effect	58%	-	58%
Induced inconsistencies as side-effect	8%	-	8%
Resolution choice reduction	93%	-	93%
Validation errors resolved via consistency check	92%	100%	93%

with the process architect during the validation step. If such a resolution was number 1 in the list of a total of 15 resolutions, then we had to examine only $1 \div 15 \times 100 = 7\%$ of the resolutions to find the appropriate one. In such a case, we say that the resolution choice reduction is $100\% - 7\% = 93\%$. The resolution choice reduction value provided in the table for Use Case 1.1 is the average over all inconsistencies that could be resolved automatically.

Finally, the last row in Table 9.2 considers those validation errors, fixing which in the object life cycle model gave rise to inconsistencies that could be subsequently resolved either automatically or manually. In the previous section, we explained that an invalid transition split does not give rise to inconsistencies. Due to this, only 11 out of 12 validation errors could be resolved via the consistency check in Use Case 1.1.

The weighted average over the use cases is computed and shown in the last column of Table 9.2. For example, the weighted average for the automatically resolved inconsistencies is computed as follows: $(92 \times 11 + 0 \times 1) \div 13 = 84\%$.

The details of the identified validation errors, detected inconsistencies and their resolution are provided in Appendix B.

9.2.3 Evaluation of Results

The results of this case study clearly imply the rejection of the null hypothesis H0. Through the validation of the extracted object life cycle models enabled by our solution, we identified validation errors and hence established that the process models created by the process architect using WBM did not correctly capture the intended object state evolution. We therefore conclude that a state-of-the-art approach is not sufficient to produce valid models that correctly capture the intended process logic and object state evolution.

Apart from locating and classifying the validation errors, the consistency check and inconsistency resolution in our solution were applied to resolve 93% of these errors. Therefore, we conclude that our solution can significantly improve the validity of models developed using a state-of-the-art approach (H1).

For the 4 inconsistencies that could be automatically resolved and were not resolved as a side-effect, the effort of the modeler in examining the alternative resolutions was reduced by 93% on average. In fact, the appropriate resolutions always appeared first in the ranked resolution lists, the length of which ranged between 12 and 15 (see Appendix B for more details). Furthermore, 58% of all the inconsistencies did not have to be examined at all, since they were resolved as a side-effect of applying resolutions to other inconsistencies. This shows that the resolution ranking based on side-effects that forms part of our

solution can significantly reduce the modeler effort required during inconsistency resolution (H3).

Overall, the case study showed that capturing the intended state evolution of objects is challenging even if the process models under consideration are not overly complex. Our experiences in this study and the feedback of the process architect suggest that modeling and validation activities should be performed in several iterations, since aligning process models with the intended object life cycles often results in adding or refining the process models. Apart from facilitating validation, the extracted object life cycle models were seen as a helpful complementary view and used in discussion with other stakeholders by the process architect.

9.3 Case Study 2: Insurance Reference Models

The IBM Insurance Application Architecture (IAA)[2] is a collection of information, process and integration models that was developed over several years to represent best-practice application development in the insurance industry. The IAA subset of interest comprises the so-called analysis-level models, namely the Analysis Process Model (APM) and the Business Object Model (BOM). APM consists of over 250 process models, offered in WBM among other formats. BOM predominantly comprises data models that capture the structure of and relationships between business objects, represented as class diagrams in the IBM Rational Software Architect (RSA). BOM additionally contains over 20 UML State Machines in RSA that represent life cycles for the main insurance objects. According to the IAA methodology, APM, BOM and other IAA models are used to generate skeletal implementation artifacts, such as BPEL, WSDL and Java code.

In the current IAA offering, there is a disconnect between the process and object life cycle models. On the one hand, actions in process models in WBM are annotated with comments in natural language containing information about states of objects at different points in the process (e.g. “Claim state becomes open”). On the other hand, object life cycle models in RSA represent the state evolution protocols for objects manipulated in the process models. While currently the object states are not used for the generation of implementation artifacts, this is a potential future requirement. However such a generation is realized, it is a prerequisite that consistency of the process model state specifications and the object life cycle models is established first.

The goal of this case study is to assess the value of our solution for attaining consistency of already existing models, as opposed to using it during the development of new models. Our general claim is that our solution enables one to achieve consistency between process and object life cycle models by locating and classifying model inconsistencies, and assisting the modeler during the resolution of these inconsistencies. We focus on the validation hypotheses H0, H2 and H3 defined in Section9.1.1.

We consider WBM and RSA to embody state-of-the-art approaches for process and object life cycle modeling. As already explained in the first case study, WBM is a commercial tool that supports process modeling with object state specifications. In turn, RSA is a commercial tool for UML modeling, hence supporting object life cycle modeling using UML State Machines. Prior to this case study, we examined a selected subset of IAA process and object life cycle models together with the domain experts from the IAA modeling team to get a sound understanding of the logic represented by these models. The models required some pre-processing before they could be taken as an input by Object Life Cycle Explorer,

[2] `http://www.ibm.com/industries/financialservices/doc/content/solution/278918103.html`

e.g. state specifications needed to be extracted from the comments in the process models. Once the models were ready, we used Object Life Cycle Explorer to detect and resolve inconsistencies in these models.

In the following, we provide an overview of the models used for the study, explain the pre-processing steps, present the gathered quantitative data, and evaluate the results.

9.3.1 Models and Model Pre-Processing

As the subject of the case study, we selected the process models that represent the handling of insurance claims and the $Claim$ and $Benefit\ In\ Claim$ object types, which are manipulated by the actions in the selected process models and at the same time are associated with object life cycle models. As shown in Table 9.3, there are 20 process models capturing claims handling in IAA. These process models contain a total of 148 actions, including atomic action and those that reference other process models. The process models form an intricate composition hierarchy, with 9 being the deepest hierarchy level. This is illustrated in Figure 9.7, where the composition hierarchy rooted by the $Administer\ Claim$ process model is partially shown. The average number of states in the two selected object types, excluding initial and final states, is 12.

Table 9.3: Model statistics for IAA

Model set	Process models	Total actions	Deepest hierarchy	Object types	States per type
Administer Claim	20	148	9	2	12

Many of the process models in the selected set have significantly complex control and data flows, involving many decisions and merges, as well as cycles, illustrated by the multitude of lines in the process overviews in Figure 9.7. All process models use routed data flow, as shown in the process extract in Figure 9.7. Comments containing object state information are relatively sparse in the process models.

Two main tasks were performed during the pre-processing of the original models: extraction of a state specification from comments in WBM and transfer of object life cycle models from RSA to Object Life Cycle Explorer. During the first task, we encountered two main types of comments attached to actions and other nodes in the process models: "t state is s_1 or ... or s_n" and "t state becomes s_1 or ... or s_n" where t is an object type and $s_1, ..., s_n$ are states of t. These two types of comments clearly correspond to accepted and produced states of nodes. Since the native state-modeling support in WBM is not well-suited for modeling multiple accepted states (see in Chapter 8), we only extracted the produced states from the comments. The process extract in Figure 9.7 shows an example of a produced state extracted from a comment, i.e. the produced state of the $Allocate\ Claim\ To\ Adjuster$ action for $Claim$ is set to $Under\_Evaluation$ based on the comment attached to this action. Additionally, sometimes there was a condition for the produced states in the comments, e.g. "When accepted, Claim state becomes Granted". Actions with such comments were generally followed by decisions, output branches of which were used to specify the alternative produced states as shown in Figure 9.8(a). Sometimes comments were attached to nodes that did not have data flows of the type in question, in which case comments were ignored.

For the second pre-processing task, transfer of object life cycle models, the object life cycles in focus were simply re-modeled using Object Life Cycle Explorer. Final states, not properly indicated in the original object life cycle models, were made explicit in the re-

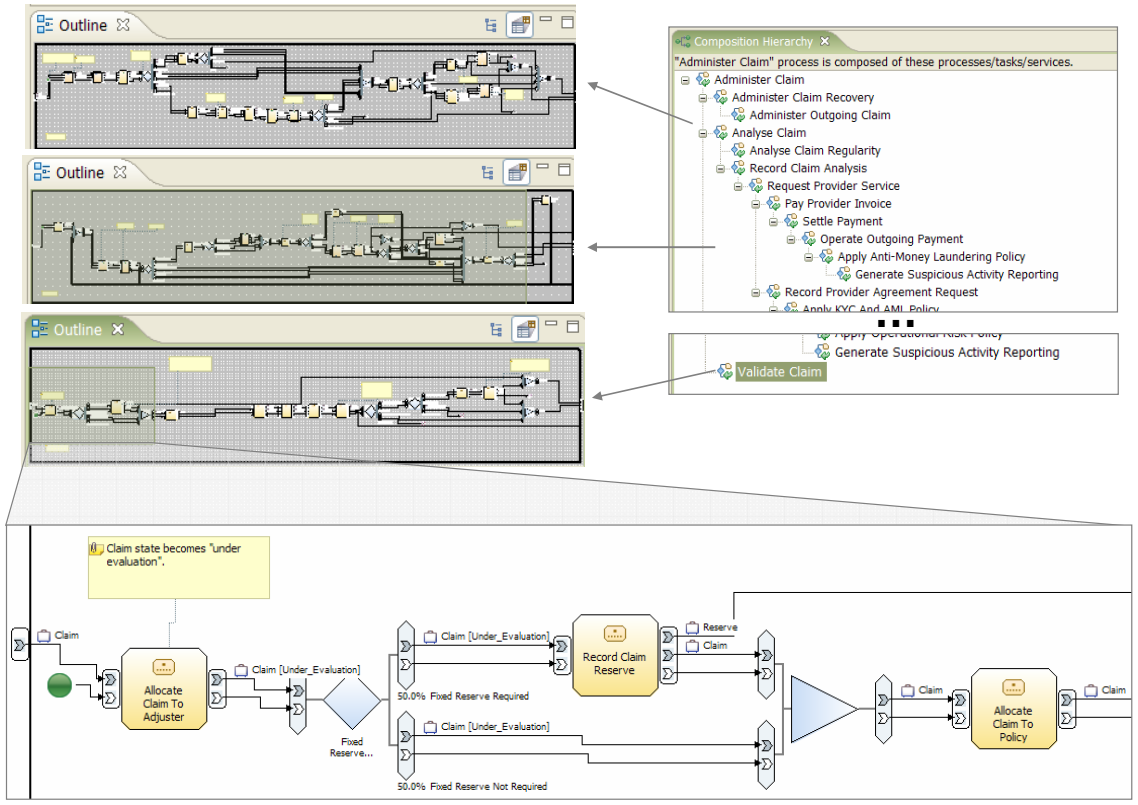


Figure 9.7: IAA Administer Claim hierarchy and example process overviews and extract

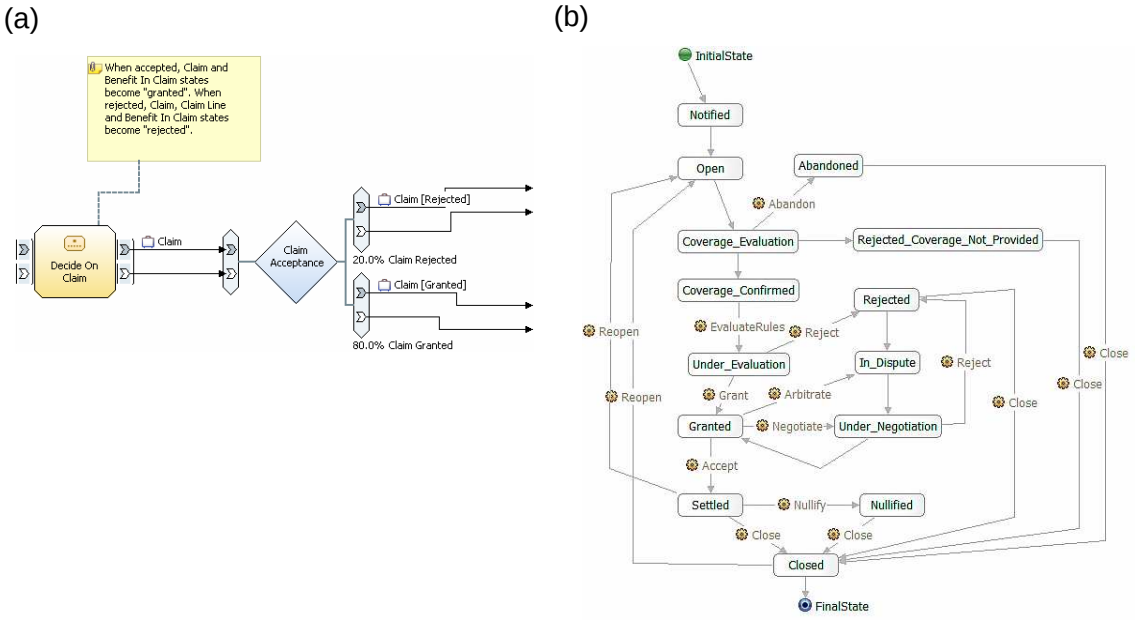


Figure 9.8: (a) Extracting produced states of a decision (b) Claim object life cycle model

created models. The largest of the two, the $Claim$ object life cycle model, is shown in Figure 9.8(b).

9.3.2 Quantitative Data

The consistency check in Object Life Cycle Explorer was performed on the $Administer\ Claim$ process model and the $Claim$ and $Benefit\ In\ Claim$ object life cycle models using the hierarchy traversal option to identify 45 inconsistencies in total, as shown in Table 9.4. The table also shows the classification of these inconsistencies according to the different inconsistency types. Further details about the inconsistencies are provided in Appendix B.

Table 9.4: Inconsistencies in IAA Administer Claim

	Claim	Benefit In Claim	Total
Non-conformant transitions	6	1	7
Non-conformant initial transitions	1	0	1
Non-conformant final transitions	3	4	7
Non-covered transitions	17	9	26
Non-covered initial transitions	1	1	2
Non-covered final transitions	0	2	2
All inconsistencies	28	17	45

During the inconsistency resolution, we confined our focus to the inconsistencies discovered with respect to the $Claim$ object life cycle model only. Since this object type has the largest object life cycle model and gave rise to the most inconsistencies, we deemed studying the resolution of these inconsistencies sufficient to provide representative results for the $Administer\ Claim$ model set. The aggregated results for inconsistency resolution are shown in Table 9.5. The number of inconsistencies shown in the table, 32, includes the 28 originally identified inconsistencies and the 4 that were induced as side-effects during the inconsistency resolution process.

As in the first case study, we resolved the inconsistencies, recording when this could be done completely automatically using Object Life Cycle Explorer and otherwise, when the manually performed resolutions were potentially automatable. In the percentage of the automatable resolutions, we also included those manual resolutions that can be partially automated. For example, some manual resolutions involved adding an action and reconnecting several edges to logically integrate the new action into the control and data flow of the process model. In some such cases, an automated resolution can be implemented to add such an action into the process model and partially connect it to the other nodes with edges, in which case we considered such a resolution automatable. The resolution choice reduction is computed as for the first case study (see Section 9.2.2).

The percentage of the inconsistencies resolved/induced as a side-effect of resolving other inconsistencies refers to side-effects of manual resolutions in this case, whereas these were side-effects of automated resolutions in the first case study. This is a direct consequence of the fact that only 3% of the inconsistencies could be resolved completely automatically in these models.

For the larger part of the inconsistency resolution, changes were made to the process models to align them with the state evolution captured in the $Claim$ object life cycle model. However, there were 2 inconsistencies, 6% of the total number, where it was more appropriate to change the object life cycle model.

Table 9.5: Inconsistency resolution results for IAA

	Claim
Inconsistencies	32
Automatically resolved inconsistencies	3%
Automatable manual resolutions	50%
Resolved inconsistencies as side-effect	59%
Induced inconsistencies as side-effect	13%
Resolution choice reduction	96%
Inconsistencies resolved by changing object life cycle model	6%

9.3.3 Evaluation of Results

The results obtained in this case study clearly imply the rejection of the null hypothesis H0. Using the consistency check in our solution, we identified 45 inconsistencies in the selected IAA model set. We therefore conclude that a state-of-the-art approach is not sufficient to develop consistent process and object life cycle models.

Our solution located and classified the inconsistencies between the process and object life cycle models. Performing this task manually would have been very challenging due to the size of the process models, their deep hierarchy, complex control and data flows, as well as a sparse state specification, which all make it difficult to see which states reach different parts of the processes. Although the resolutions currently implemented in Object Life Cycle Explorer could only be applied to automatically resolve 3% of the studied inconsistencies, 50% of the manual resolutions can potentially be implemented to extend Object Life Cycle Explorer. Therefore, our solution has the potential to automatically resolve 53% of the studied inconsistencies. Hence, we conclude that our solution can improve the consistency of models developed using a state-of-the-art approach (H2).

The resolution choice reduction value obtained was 96%, however this value is based on the alternative resolutions available for the only inconsistency that could be resolved automatically. Even though this value is high, it is not sufficient to reliably conclude that our solution significantly reduces the modeler effort during inconsistency resolution (H3).

On the whole, the case study demonstrated that without the adequate support for integration of process and object life cycle modeling, inconsistencies arise even in very mature models developed over many years. During the case study, the complexity of the models often made it quite difficult and time-consuming to understand the reason for a particular inconsistency and how it should be resolved. We believe that these difficulties could have been partially avoided if the models were in the first place developed using an iterative approach alternating between modeling, consistency checking and inconsistency resolution. The task of examining the inconsistency sources itself could be alleviated by an additional visualization of the computed states for all data-flow edges, which would make it easier for the modeler to see the reachable states at each point in a given process model.

In this case study, we considered only a small subset of the IAA process and object life cycle models, which already lead to a significant amount of inconsistencies. We handled the inconsistencies in the order that they were displayed by Object Life Cycle Explorer. Since this was an arbitrary order, we had to perform many context-switches while handling these inconsistencies. If the inconsistency prioritization described in Chapter 5 was already implemented in Object Life Cycle Explorer, our task in handling and traversing these inconsistencies would have been easier. Such prioritization would be even more

crucial if a larger subset of the IAA models were considered.

9.4 Threats to Validity

In any empirical research, it is important to consider the possible threats to the validity of the study. In this context, two main types of validity are generally distinguished: internal and external validity [Campbell and Stanley, 1963]. *Internal validity* is concerned with the reliability of the causal relationship inferred from a particular study. On the other hand, *external validity* relates to the generalization of a particular finding established in the study at hand.

Possible threats to the internal validity of our results include the proficiency of modelers involved in the case studies, the choice of tools considered to be the state of the art and our own participation in some of the case study activities.

In both case studies, experienced modelers were responsible for the creation of the study models. In the CAD Management case study, it was one process architect, whose daily responsibilities include process modeling with WBM. In the Insurance Reference Models case study, the process and object life cycle models were created by a team of modelers, who have been using WBM and RSA for several years. Therefore, the discovered validation errors and inconsistencies cannot be attributed to the incompetence of the modelers.

WBM and RSA are among the leading tools for process and software modeling, which leads us to conclude that it is fair to refer to them as state-of-the-art. We are not currently aware of any other tools that explicitly address the integration of process and object life cycle modeling. However, it is possible that tools that provide different visualization or method support for process and object life cycle modeling could lead to fewer validation errors and inconsistencies being detected in our case studies.

Our own participation in some of the case study activities was necessary to ensure that quantitative data about the case studies is properly recorded. In both case studies, we did not contribute at all to the creation of models in question. Our participation in the subsequent activities could not influence the validation errors and inconsistencies inherent in the created models.

With regards to the external validity of our results, we have to consider how representative our case studies are for the general case. The project under consideration in the CAD Management case study exhibits requirements that are typical to organizations in many different industries today. The particular domain of this case study does not play a significant role, since the structure of the process models and the type of object state evolution are very generic. The IAA insurance reference models considered in the second case study are similar to reference model collections that exist in other industries (e.g. HL7[3], IFW[4]). Some of these do not currently contain both process and object life cycle models, but many of them are still under development and may evolve to include these models in the future. Therefore, we conclude that there is no significant threat to the generalization of our case study results.

[3] http://www.hl7.org

[4] http://www-03.ibm.com/industries/financialservices/doc/content/solution/391981103.html

9.5 Summary and Discussion

In this chapter, we have demonstrated that our solution surpasses state-of-the-art approaches in obtaining process and object life cycle models that are valid and consistent. Object Life Cycle Explorer and its method have proven feasible and effective not only for achieving consistency of existing process and object life cycle models, but also when applied during the development of new models in a live project.

In future, it would be valuable to complement the two case studies presented here with a survey across a number of similar cases. Using different state-of-the-art tools would also aid in making our obtained results more reliable and generalizable.

Chapter 10

Conclusion

In this dissertation, we have presented our proposed framework for the integration of process and object life cycle modeling in the context of Business Process Management (BPM). We have described the tool support and developed to realize this framework. By means of two case studies, we have validated our solution.

In this final chapter, we conclude the dissertation by summarizing the main contributions in Section 10.1, discussing the broader impact of the dissertation in Section 10.2, providing an outlook on possible future research in Section 10.3 and reflecting on our overall insights about BPM in Section 10.4.

10.1 Summary of Contributions

The problem addressed in this dissertation was the integration of process and object life cycle models, which constitute two different views during the development of applications in the context of Business Process Management (BPM). While process models focus on the flow of control and business objects between the tasks of a business process, object life cycle models focus on the overall state evolution of business objects.

As a solution to the problem, we have presented a framework for integrated modeling of processes and object life cycles. This framework and the main concepts pertaining to each of its components are shown in Figure 10.1. In the following, we summarize the contributions for each framework component and emphasize their importance.

1. **Syntax and semantics of process and object life cycle models:** We formalized the syntax and execution semantics of data flow and object state specifications in process models. Two types of data flow were explicitly distinguished: repository data flow with pass-by-reference semantics and routed data flow with pass-by-value semantics. A formal semantics was defined to capture the difference of object state evolution in process models with different types of data flow. For object life cycle models, we defined the notions of object life cycle conformance and coverage that allow one to use an object life cycle model as a protocol of object state evolution.

   Although data flow is an essential aspect of process modeling, there is currently no agreement about a particular representation or interpretation of data flow in process models. Differences between routed and repository representations, and pass-by-value and pass-by-reference semantics have been described informally in the existing literature [Sadiq et al., 2004, Russell et al., 2005]. Our work provides the first consolidated formalization of these concepts.

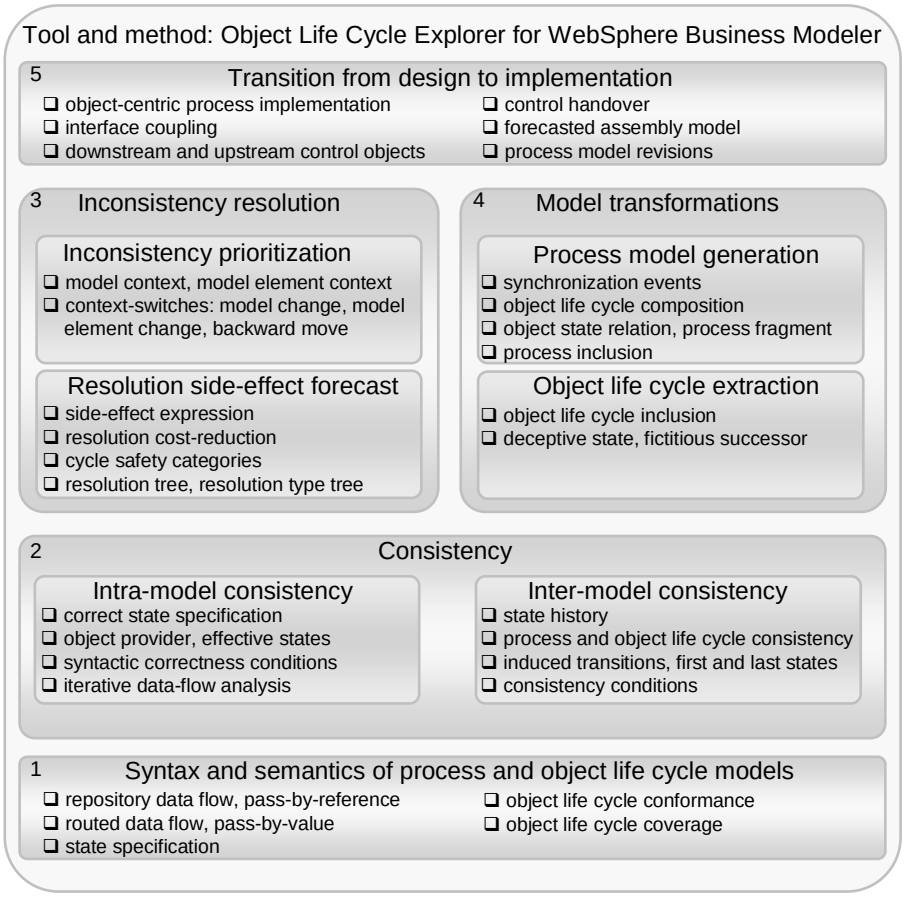


Figure 10.1: Main concepts of proposed solution

Specification of object states in process models has proved to be a valuable aspect of process modeling languages [UML, 2007b, BPMN, 2008]. However, the semantics of such a specification is scarcely described. We provide a precise semantic definition for object state specifications, which can be used to enhance existing process modeling languages and as a basis for further research.

Object life cycle models have been given execution semantics in several existing works [Kappel and Schrefl, 1991, van der Aalst and Basten, 2001]. This type of semantics cannot be used to directly interpret an object life cycle model as a protocol of object state evolution. In contrast, our proposed notions of object life cycle conformance and coverage are devised for the purpose of describing an object life cycle model as a state evolution protocol.

2. **Consistency:** For intra-model consistency of process models, we provided a semantic definition of the correctness of process models with object state specifications. We defined syntactic correctness conditions based on the concepts of object providers and effective states, which can be evaluated using a static analysis technique based on iterative data-flow analysis. For inter-model consistency of process and object life cycle models, we provided a definition based on the semantic domain of object state sequences. We defined syntactic consistency conditions using the concepts of induced transitions, and first and last states of a process model. These conditions can also be statically evaluated based on iterative data-flow analysis.

The importance of correctness and verification of process models has been emphasized in much of the existing literature (e.g. [van der Aalst, 1997, Sadiq et al., 2004, Vanhatalo et al., 2007]). Our definition of state specification correctness complements the existing control-flow and data-related correctness notions.

For multi-view modeling, a clear definition of consistency has been identified as crucial [Küster, 2004, Dijkman, 2006, van Hee et al., 2006]. Our work provides the first precise definition of process and object life cycle model consistency. This definition can be used as a benchmark by any approaches that use these two types of models as complementary views.

3. **Inconsistency resolution:** We presented an approach to developing inconsistency resolutions with side-effect expressions, which facilitates the forecast of resolution side-effects and the analysis of resolution safety with respect to cycles. By associating inconsistencies with costs, the overall cost-reduction of resolutions is computed based on their side-effect forecasts and used to rank alternative resolutions. Furthermore, we proposed to prioritize detected inconsistencies by grouping and ordering them in a way that the number of context-switches required by the modeler during inconsistency resolution is minimized.

   Existing work on inconsistency resolution side-effects and cycles computes side-effects and cycles that can potentially occur during resolution of inconsistencies in a concrete model [Mens and Van Der Straeten, 2006, Egyed, 2007]. Our approach supports the forecast of the exact resolution side-effects for a given model.

   Supporting the modeler in selecting among alternative resolutions and in handling large numbers of inconsistencies has been identified as crucial for inconsistency management in [Spanoudakis and Zisman, 2001]. As shown by the validation, our approach of using side-effects to rank alternative resolutions significantly reduces the effort and time required for the identification of the best resolution. Additionally, our approach to inconsistency prioritization assists the user in handling large sets of inconsistencies.

4. **Model transformations:** We developed two techniques, namely the object life cycle extraction and process model generation, for transforming process models to object life cycle models and vice versa. We showed that both techniques ensure consistency of the source and target models, and additionally demonstrated that the target models are minimal in the set of all models consistent with the source model. Object life cycle extraction uses induced transitions, first and last states computed for a given process model to construct an object life cycle model for each manipulated object type. Process model generation involves manual identification of synchronization events in the given set of object life cycle models, after which an object life cycle composition is computed and used to construct the final process model.

   Model transformations are generally considered essential in Model-Driven Engineering (MDE) [Kleppe et al., 2003, Sendall and Kozaczynski, 2003] and multi-view modeling [Küster, 2004, Van Der Straeten, 2005]. In the context of the integration of process and object life cycle models as complementary views, transformations between these two model types enable the Validate-Check-Resolve and Reference-Driven Process Modeling strategies presented in Chapter 8. In one of the case studies, we demonstrated that the Validate-Check-Resolve strategy is essential in validating object state evolution captured by state specifications in process models.

5. **Transition from design to implementation:** In support of the transition from process modeling to an object-centric process implementation, we presented a technique for managing coupling of object life cycle components. Based on an informal description of a mapping between process models and object life cycle components, we identified the so-called control handovers and decision notifications as the two main sources of coupling. To enable coupling management already during the design phase, we developed a technique for computing the expected coupling of object life cycle components by statically analyzing a given process model.

   Several object-centric approaches to process implementation have emerged in the recent years (e.g. [van der Aalst et al., 2001, Nandi and Kumaran, 2005, Müller et al., 2007]). To facilitate the realization of such approaches, a fully behavior-preserving transformation from process models to object life cycle components needs to be developed. Furthermore, various issues arising during the transition from design to implementation have to be carefully studied. Our work explored such a behavior-preserving transformation and addressed coupling, as one important issue arising in this context. Our results can be used as a starting point in an extensive study devoted to facilitating the derivation of object-centric process implementations.

10.2 Impact on BPM and MDE

We now review the broader impact of the work detailed in this dissertation on the fields of BPM and MDE. In the following, we discuss the impact with regards to the concepts, modeling languages, tools and methods established in these fields.

10.2.1 Concepts

Adoption of a new concept by researchers and practitioners in a particular field usually happens gradually, requiring several seminal works to describe the concept and demonstrate its value. Although there is generally no complete overlap between the concepts used in research and those used in practice, a large portion of the concepts should be common to both, especially in practice-driven fields like BPM and MDE.

This dissertation contributes to the understanding of a range of concepts related to BPM and MDE. Many of these, such as *inconsistency*, *model transformation* and *coupling*, are already well-established, at least among researchers. However, there are four particular concepts extensively studied in this work that have not yet achieved adequate attention and understanding in the field. We discuss these next, together with their broader application.

- *Object state:* Object states, as they are used in this dissertation, seem to be a natural abstraction for intermediate milestones reached during the overall processing of an object. Many participants and stakeholders of business processes already use object states to define their goals or report their progress with regards to some task (e.g. $Deal\ Closed$, $Contract\ Reviewed$). Despite this evidently widespread use of object states in the BPM context, this concept has not been given much attention in the existing literature. Our work studies the concept of an object state in detail and precisely defines how this concept relates to other concepts that are already established in BPM, such as *process model* and *data flow*. We therefore believe that this work can serve as a foundation for establishing object state as a prominent concept

in the BPM field, which would facilitate the explicit consideration of object states in upcoming modeling languages, tools and methods.

- *Object life cycle:* The concept of an object life cycle is well-known in object-oriented modeling, but is not yet a key concept in BPM. Several recent research publications in the BPM field make use of object life cycles (e.g. [van der Aalst et al., 2001, Nigam and Caswell, 2003, Müller et al., 2006, Bhattacharya et al., 2007]), which indicates that this is an emerging concept in the research community. Our work further contributes to the understanding of this concept and by making Object Life Cycle Explorer available on IBM alphaWorks, we facilitate the dissemination of this concept into practice. Practitioner interest in the concept of an object life cycle is stated in the article about Object Life Cycle Explorer posted on InfoQ[1], an independent online forum for enterprise software developers.

- *Resolution side-effect:* In the case studies, we demonstrated the potential of using the information about the expected side-effects of inconsistency resolutions to reduce the effort required by the modeler while resolving inconsistencies. Although the concept of a side-effect has already been identified in some research literature on MDE [Mens et al., 2006a, Mens et al., 2006b, Egyed, 2007], we have shown a new aspect that contributes to the importance of this concept.

- *Context-switching:* As part of our inconsistency resolution approach, we propose to prioritize inconsistencies such that context-switching is minimized for the modeler resolving the inconsistencies. To the best of our knowledge, the concept of context-switching does not appear in the existing literature in the field of MDE. Since automated support for correction of inconsistencies or errors is still not widespread in the existing modeling tools, the significance of context-switching in this context is still to be determined. Our work can be used as a first reference to the use of this concept in the MDE field.

10.2.2 Languages

In this dissertation, we clarified the semantics of data flow and state specifications in process models and defined the consistency for process and object life cycle models. The semantic clarification can be directly integrated into existing process modeling languages, such as UML Activity Diagrams and BPMN. The consistency definitions can be used to extend the UML specification to define the relation between UML Activity Diagrams and UML State Machines. This extension would however cover a subset of UML Activity Diagrams and UML State Machines, since there are some constructs that we do not consider in our definitions of process and object life cycle models.

By establishing the relation between process and object life cycle models, we also provide a link from process modeling to object-oriented modeling. The potential benefits of bridging these two modeling paradigms have been highlighted by Loos et al [Loos and Allweyer, 1998, Loos and Fettke, 2001], but since then, not much progress has been achieved on this topic. Our work can be extended to define the relation between process models and other object-oriented models, such as class and sequence diagrams. Bridging the gap between these paradigms would also influence the related modeling languages.

[1] `http://www.infoq.com/news/2008/06/olc-wbm`

10.2.3 Tools

Our framework for integrated process and object life cycle modeling can be easily incorporated into an existing process modeling tool. We demonstrated this by implementing Object Life Cycle Explorer as an extension to IBM WebSphere Business Modeler. Incorporating the framework into a tool that is based on other process or object life cycle modeling languages is not a challenge, since the framework is built on generic process and object life cycle modeling concepts.

Capabilities of tools such as IBM WebSphere Business Modeler can be further enriched to leverage the integration of process and object life cycle modeling. We primarily see potential in extending process simulation and monitoring to allow one to directly simulate and analyze the achievement of milestones related to object states. By establishing a relation between key performance indicators and object states, the state evolution of objects and its effect on the overall business performance could be directly monitored during process enactment.

In our work, we have primarily focused on the *process architect* as the target user of tools like Object Life Cycle Explorer (cf. CAD Management case study in Chapter 9). We envision that in the future, the relation between process modeling and object-oriented modeling established in our work would also facilitate more coordination between process architects and project roles responsible for object-oriented modeling.

10.2.4 Methods

As demonstrated by our two case studies, the capabilities of Object Life Cycle Explorer can be employed in several different ways to improve model quality. In Chapter 8, we described three modeling strategies to capture the use of Object Life Cycle Explorer in different scenarios. These modeling strategies can be integrated into a broader method used during the analysis and design phases of the BPM life cycle.

Methods to support the transition from the design phase to the implementation phase of the BPM life cycle are still under research. While many mappings from process modeling to process implementation languages have been proposed (e.g. [Ouyang et al., 2006, Recker and Mendling, 2006]), the design-to-implementation transition is bound to be much more than invoking a generation of a process implementation from a process model. In Chapter 7, we have shown that coupling needs to be managed when an object-oriented approach to process implementation is employed. We believe that concerns such as high coupling need to be addressed by explicit method steps that occur before and after the derivation of a process implementation.

The results of this dissertation can also be leveraged in contexts other than the analysis and design of business processes. This is further discussed in the following section on future research.

10.3 Future Research

We envision several directions for how the contributions of this dissertation could be extended in the future. As we describe next, some extensions would be direct enhancements of our proposed solution, while others would explore the value of our solution in new contexts.

In the work described in this dissertation, we focused on the most fundamental process modeling constructs and did not consider some more advanced constructs for control-flow and data-flow modeling. It would be valuable to extend our work to cater for other process modeling constructs, which are often required in practice. For example, some scenarios require the execution of the same business process task several times, once for each object in a given collection. In workflow patterns, such behavior is described as the so-called multiple instance patterns [van der Aalst et al., 2003]. If constructs that represent such behavior are added to process models considered in this work, the effects on the correctness of state specifications and consistency with object life cycle models would need to be explored.

Our proposed evaluation methods for correctness of state specifications and consistency of process and object life cycle models may not produce accurate results for process models that contain forks and are associated with repository data flow. In our case studies, we did not encounter such types of process models. However, to accommodate for precise results being computed for all possible process models, the application of other existing approaches to static analysis of process models (e.g. [Vanhatalo et al., 2007]) could be investigated.

Our results on the transition from process modeling to object-centric process implementations address only a selection of issues in this space. We believe that more work devoted to studying this transition is needed to facilitate a wider adoption and acknowledgement of object-centric approaches. Furthermore, criteria for comparing activity-centric and object-centric process implementation approaches should be defined and used to draw an objective evaluation of the similarities and differences between these approaches.

Apart from the direct enhancement of our work described above, it would also be interesting to investigate the extension of our proposed solution on a broader scale. For instance, we believe that object life cycle models could play an essential role in integrating the specification of data-related security aspects into the BPM activities. Currently, most literature in the BPM area does not explicitly take security considerations into account. However, role-based access control has to be specified in this context as for most other applications. Since access rights to the data of a particular business objects can vary depending on the state of the object, object life cycle models present a convenient basis for specifying these rights. Some work in this direction outside the BPM area is described as data-centric security [Sreedhar, 2006].

Finally, it would be interesting to investigate the relationship between object state specifications in process models and the domain of semantic web services [McIlraith et al., 2001]. Specification of accepted and produced states for business process tasks correspond to pre-conditions and effects that form a semantic description of a service. By mapping objects and their states to standardized ontologies [Gruber, 1995], object state specifications can be directly used during the dynamic binding of tasks to services. On the other hand, object life cycle models could be used to represent stateful specification of services. Further investigations are required to determine whether or not it is sufficient to use state specifications to describe services and meaningful to derive services based on object life cycle models.

10.4 Concluding Remarks

As our final concluding remarks, we point out some of our general insights about Business Process Management (BPM) obtained during the course of this work.

Despite the promise of MDE approaches to simplify software development, modeling does not remove the complexity intrinsic in the applications under development. In the context of BPM, this is witnessed by the overly complex languages being developed to model business processes. It takes hundreds of pages to specify languages such as UML Activity Diagrams and BPMN, excluding any precise formulation of their semantics. Such complexity makes these languages inaccessible to domain experts and business analysts responsible for process modeling, leading to incorrect or invalid process models being developed. In our view, a radical simplification of process modeling languages and comprehensive method support are required to enable effective process modeling.

Above all, we see representation of data flow in process models as a particularly weak point in process modeling languages of today. The semantic ambiguities of data flow discussed in this dissertation constitute one of the problems. In the course of examining large collections of best practice process models for the insurance and banking domains, we identified several other issues. Many of the current graphical representations of data flow drastically clutter process models and deteriorate their understandability. Furthermore, there is a lack of method support for refining data in a process model during the transition from the analysis phase to the design phase. We hence conclude that an ideal representation and method for capturing data in process models is still to be identified.

The general concept of transforming process models to an implementation introduces some problems into the vision of the BPM life cycle, irrespective of whether an activity-centric, object-centric or another approach to implementation is used. There is still no comprehensive solution for keeping models at the design level and the implementation of these models synchronized. Since manual refinements of a generated implementation are usually required, the implementation cannot be simply regenerated once changes occur at the design level, as envisioned in the BPM life cycle. Therefore, the continuous improvement depicted by the loop connecting evaluation to design and design to implementation is still a vision rather than the state of the practice.

It is therefore evident that a number of open gaps still remain in the BPM field. However, we believe that the gaps such as the ones described above will be closed as the BPM field matures, taking the meaning of business-IT integration to a new dimension.

Bibliography

[Agrawal, 2004] Agrawal, A. (2004). *A Formal Graph Transformation Based Language for Model-to-Model Transformations*. Dissertation for PhD in Electrical Engineering, Faculty of the Graduate School of Vanderbilt University, Nashville, Tennessee.

[Agrawal et al., 2004] Agrawal, A., Simon, G., and Karsai, G. (2004). Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. *Electronic Notes in Theoretical Computer Science*, 109:43–56.

[Alur and Dill, 1992] Alur, R. and Dill, D. L. (1992). The Theory of Timed Automata. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 45–73. Springer.

[Ambler and Jeffries, 2002] Ambler, S. W. and Jeffries, R. (2002). *Agile Modeling: Effective Practices for EXtreme Programming and the Unified Process*. J. Wiley.

[Arnold, 1994] Arnold, A. (1994). *Finite Transition Systems: Semantics of Communicating Systems*. Prentice Hall International (UK) Ltd. Translated by J. Plaice.

[Ashworth and Goodland, 1989] Ashworth, C. and Goodland, M. (1989). *SSADM: A Practical Approach*. McGraw-Hill.

[Backus et al., 1963] Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., van Wijngaarden, A., Woodger, M., and Naur, P. (1963). Revised Report on the Algorithm Language ALGOL 60. *Communications of the ACM*, 6(1):1–17.

[Badouel and Darondeau, 1998] Badouel, E. and Darondeau, P. (1998). Theory of Regions. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, 1996*, volume 1491 of *LNCS*, pages 529–586. Springer.

[Becker et al., 2007] Becker, S., Herold, S., Lohmann, S., and Westfechtel, B. (2007). A Graph-Based Algorithm for Consistency Maintenance in Incremental and Interactive Integration Tools. *Software and System Modeling*, 6(3):287–315.

[Beers and Carey, 2006] Beers, G. and Carey, J. (2006). WebSphere Process Server Business State Machines Concepts and Capabilities, Part 1: Exploring Basic Concepts. IBM developerWorks.

[Bhaduri and Venkatesh, 2002] Bhaduri, P. and Venkatesh, R. (2002). Formal Consistency of Models in Multi-View Modelling. In *Consistency Problems in UML-based Software Development: Workshop Materials, Research Report 2002-06, Blekinge Institute of Technology*, pages 149–159.

[Bhattacharya et al., 2007] Bhattacharya, K., Gerede, C., Hull, R., Liu, R., and Su, J. (2007). Towards Formal Analysis of Artifact-Centric Business Process Models. In *Proceedings of the 5th International Conference on Business Process Management (BPM 2007)*, volume 4714 of *LNCS*, pages 288–304. Springer.

[Bhattacharya et al., 2005] Bhattacharya, K., Guttman, R., Lyman, K., III, F. F. H., Kumaran, S., Nandi, P., Wu, F. Y., Athma, P., Freiberg, C., Johannsen, L., and Staudt, A. (2005). A Model-Driven Approach to Industrializing Discovery Processes in Pharmaceutical Research. *IBM Systems Journal*, 44(1):145–162.

[Bock, 2003a] Bock, C. (2003a). UML 2 Activity and Action Models. *Journal of Object Technology*, 2(4):43–53.

[Bock, 2003b] Bock, C. (2003b). UML 2 Activity and Action Models Part 2: Actions. *Journal of Object Technology*, 2(5):41–56.

[Bock, 2003c] Bock, C. (2003c). UML 2 Activity and Action Models Part 3: Control Nodes. *Journal of Object Technology*, 2(6):7–23.

[Bock, 2004] Bock, C. (2004). UML 2 Activity and Action Models Part 4: Object Nodes. *Journal of Object Technology*, 3(1):27–41.

[Booch, 1994] Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*. Benjamin-Cummings Publishing Co., Inc., 2nd edition.

[BPEL, 2003] BPEL (2003). Business Process Execution Language for Web Services, Version 1.1. Joint specification by BEA, IBM, Microsoft, SAP and Siebel Systems.

[BPMN, 2007] BPMN (2007). Business Process Modeling Notation (BPMN) 2.0, Request For Proposal, BMI/2007-06-05. OMG Document.

[BPMN, 2008] BPMN (2008). Business Process Modeling Notation Specification, V1.1, formal/2008-01-17. OMG Document.

[Brand and Zafiropulo, 1983] Brand, D. and Zafiropulo, P. (1983). On Communicating Finite-State Machines. *Journal of the ACM (JACM)*, 30(2):323–342.

[Briand et al., 1999] Briand, L. C., Daly, J. W., and Wüst, J. (1999). A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25(1):91–121.

[Campbell and Stanley, 1963] Campbell, D. T. and Stanley, J. C. (1963). *Experimental and Quasi-Experimental Designs for Research*. Houghton Mifflin.

[Chen, 1976] Chen, P. P. (1976). The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36.

[Cimatti et al., 2002] Cimatti, A., Clarke, E. M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. (2002). NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*, pages 359–364. Springer.

[Clarke et al., 1999] Clarke, E., Grumberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.

[Coad and Yourdon, 1991] Coad, P. and Yourdon, E. (1991). *Object-Oriented Analysis*. Yourdon Press, 2nd edition.

[Csertán et al., 2002] Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., and Varró, D. (2002). VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE'02)*, pages 267–270. IEEE Computer Society.

[Czarnecki and Helsen, 2003] Czarnecki, K. and Helsen, S. (2003). Classification of Model Transformation Approaches. In online proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architectures.

[Czarnecki and Helsen, 2006] Czarnecki, K. and Helsen, S. (2006). Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–646.

[Date, 2000] Date, C. J. (2000). *Introduction to Database Systems*. Addison-Wesley, 7th edition.

[Desel and Esparza, 1995] Desel, J. and Esparza, J. (1995). *Free Choice Petri Nets*. Cambridge University Press.

[Dijkman, 2006] Dijkman, R. M. (2006). *Consistency in Multi-Viewpoint Architectural Design*. PhD thesis, University of Twente.

[Dijkman et al., 2008] Dijkman, R. M., Dumas, M., and Ouyang, C. (2008). Semantics and Analysis of Business Process Models in BPMN. *Information and Software Technology*, 50(12):1281–1294.

[Dumas and ter Hofstede, 2001] Dumas, M. and ter Hofstede, A. H. M. (2001). UML Activity Diagrams as a Workflow Specification Language. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML'01)*, volume 2185 of *LNCS*, pages 76–90. Springer.

[Dumas et al., 2005] Dumas, M., van der Aalst, W. M., and ter Hofstede, A. H. (2005). *Process-Aware Information Systems: Bridging People and Software Through Process Technology*. John Wiley & Sons, Inc.

[Ebbinghaus et al., 1996] Ebbinghaus, H.-D., Flum, J., and Thomas, W. (1996). *Mathematical Logic*. Springer, 2nd edition.

[Ebert and Engels, 1997] Ebert, J. and Engels, G. (1997). Specialization of Object Life Cycle Definitions. Fachberichte Informatik 19/95, University of Koblenz-Landau.

[Egyed, 2007] Egyed, A. (2007). Fixing Inconsistencies in UML Design Models. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 292–301. IEEE Computer Society.

[Ehrenfeucht and Rozenberg, 1989] Ehrenfeucht, A. and Rozenberg, G. (1989). Partial (Set) 2-Structures. Part I: Basic Notions and the Representation Problem. *Acta Informatica*, 27(4):315–342.

[Ehrig et al., 1999] Ehrig, H., Rozenberg, G., Engels, G., and Kreowski, H.-J., editors (1999). *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Theory*. World Scientific.

[Erl, 2005] Erl, T. (2005). *Service-Oriented Architecture : Concepts, Technology, and Design*. Prentice Hall.

[Eshuis, 2002] Eshuis, R. (2002). *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, University of Twente.

[Eshuis and Wieringa, 2004] Eshuis, R. and Wieringa, R. (2004). Tool Support for Verifying UML Activity Diagrams. *IEEE Transactions on Software Engineering*, 30(7):437–447.

[Favre, 2008] Favre, C. (2008). Algorithmic Verification of Business Process Models. Master's thesis, Ecole Polytechnique Federale de Lausanne.

[Finkelstein et al., 1992] Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., and Goedicke, M. (1992). Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–57.

[Finkelstein et al., 1996] Finkelstein, A., Spanoudakis, G., and Till, D. (1996). Managing Interference. In *Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints'96) on SIGSOFT'96 Workshops*, pages 172–174. ACM.

[Förster et al., 2007] Förster, A., Engels, G., Schattkowsky, T., and Van Der Straeten, R. (2007). Verification of Business Process Quality Constraints Based on Visual Process Patterns. In *Proceedings of the 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 197–208. IEEE Computer Society.

[Gamma and Beck, 2003] Gamma, E. and Beck, K. (2003). *Contributing to Eclipse: Principles, Patterns, and Plug-ins*. Addison-Wesley.

[Gantt, 1919] Gantt, H. L. (1919). *Organizing for Work*. Harcourt, Brace & Howe.

[Giese and Wagner, 2006] Giese, H. and Wagner, R. (2006). Incremental Model Synchronization with Triple Graph Grammars. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)*, volume 4199 of *LNCS*, pages 543–557. Springer.

[Giese and Wagner, 2008] Giese, H. and Wagner, R. (2008). From Model Transformation to Incremental Bidirectional Model Synchronization. *Software and Systems Modeling*.

[Große-Rhode, 2001] Große-Rhode, M. (2001). Integrating Semantics for Object-Oriented System Models. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP'01)*, pages 40–60, London, UK. Springer.

[Gruber, 1995] Gruber, T. R. (1995). Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *International Journal of Human-Computer Studies*, 43(5-6):907–928.

[Grundy et al., 1998] Grundy, J., Hosking, J., and Mugridge, W. B. (1998). Inconsistency Management for Multiple-View Software Development Environments. *IEEE Transactions on Software Engineering*, 24(11):960–981.

[Harel and Politi, 1998] Harel, D. and Politi, M. (1998). *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill.

[Hauser and Koehler, 2004] Hauser, R. and Koehler, J. (2004). Compiling Process Graphs into Executable Code. In *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE'04)*, volume 3286 of *LNCS*, pages 317–336. Springer.

[Hay, 2006] Hay, D. C. (2006). *Data Model Patterns: A Metadata Map*. Morgan Kaufmann, 1st edition.

[Heckel et al., 2002] Heckel, R., Küster, J., and Taentzer, G. (2002). Towards Automatic Translation of UML Models into Semantic Domains. In *Proceedings of APPLIGRAPH Workshop on Applied Graph Transformation (AGT'02)*, pages 11–22.

[Hermann et al., 2008] Hermann, F., Ehrig, H., and Taentzer, G. (2008). A Typed Attributed Graph Grammar with Inheritance for the Abstract Syntax of UML Class and Sequence Diagrams. *Electronic Notes in Theoretical Computer Science*, 211:261–269.

[Hollingsworth, 2004] Hollingsworth, D. (2004). *Workflow Handbook 2004*, chapter The Workflow Reference Model: 10 Years On. Future Strategies Inc.

[Holzmann, 2003] Holzmann, G. J. (2003). *SPIN Model Checker, The: Primer and Reference Manual*. Addison-Wesley.

[Hopcroft et al., 2006] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation*. Pearson Education, 3rd edition.

[Huzar et al., 2005] Huzar, Z., Kuzniarz, L., Reggio, G., and Sourrouille, J.-L. (2005). Consistency Problems in UML-Based Software Development. In *UML Modeling Languages and Applications, 2004 UML Satellite Activities, Revised Selected Papers*, volume 3297 of *LNCS*, pages 1–12. Springer.

[Jackson, 1983] Jackson, M. (1983). *System Development*. Prentice-Hall.

[Janssen and Mateescu, 1998] Janssen, W. and Mateescu, R. (1998). Verifying Business Processes using SPIN. In *Proceedings of the 4th International SPIN Workshop*, pages 21–36.

[Janssen et al., 1999] Janssen, W., Mateescu, R., Mauw, S., Fennema, P., and van der Stappen, P. (1999). Model Checking for Managers. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *LNCS*, pages 92–107. Springer.

[Jensen, 1995] Jensen, K. (1995). *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. Springer.

[Kam and Ullman, 1976] Kam, J. B. and Ullman, J. D. (1976). Global Data Flow Analysis and Iterative Algorithms. *Journal of the ACM*, 23(1):158–171.

[Kappel and Schrefl, 1991] Kappel, G. and Schrefl, M. (1991). Object/Behavior Diagrams. In *Proceedings of the 7th International Conference on Data Engineering*, pages 530–539. IEEE Computer Society.

[Karsai et al., 2003] Karsai, G., Agrawal, A., Shi, F., and Sprinkle, J. (2003). On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *The Journal of Universal Computer Science*, 9(11):1296–1321.

[Keller et al., 1992] Keller, G., Nüttgens, M., and Scheer, A. (1992). Semantische Prozessmodellierung auf der Grundlage Ereignisgesteuerter Processketten (EPK). Technical report, Veröffentlichungen des Instituts für Wirtschaftsinformatik, University of Saarland, Saarbrücken.

[Kent, 2002] Kent, S. (2002). Model Driven Engineering. In *Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM'02)*, pages 286–298. Springer.

[Kildall, 1973] Kildall, G. A. (1973). A Unified Approach to Global Program Optimization. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 194–206.

[Kilov, 1990] Kilov, H. (1990). From Semantic to Object-Oriented Data Modeling. In *Proceedings of the 1st International Conference on Systems Integration (ICSI'90)*, pages 385–393. IEEE Computer Society.

[Kindler, 2006] Kindler, E. (2006). On the Semantics of EPCs: Resolving the Vicious Circle. *Data & Knowledge Engineering*, 56(1):23–40.

[Kitchenham et al., 1995] Kitchenham, B., Pickard, L., and Pfleeger, S. L. (1995). Case Studies for Method and Tool Evaluation. *IEEE Software*, 12(4):52–62.

[Kleppe and Warmer, 2003] Kleppe, A. and Warmer, J. (2003). *The Object Constraint Language. Second Edition.* Addison-Wesley.

[Kleppe et al., 2003] Kleppe, A., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley Professional, 1st edition.

[Knuth, 1971] Knuth, D. E. (1971). An Empirical Study of FORTRAN Programs. *Software - Practice and Experience (SPE)*, 1(2):105–133.

[Koehler et al., 2003] Koehler, J., Hauser, R., Kapoor, S., Wu, F. Y., and Kumaran, S. (2003). A Model-Driven Transformation Method. In *Proceedings of the 7th International Enterprise Distributed Object Computing Conference (EDOC'03)*, pages 186–197. IEEE Computer Society.

[Kruchten, 2004] Kruchten, P. (2004). *The Rational Unified Process: An Introduction*. Addison-Wesley.

[Kumaran et al., 2008] Kumaran, S., Liu, R., and Wu, F. Y. (2008). On the Duality of Information-Centric and Activity-Centric Models of Business Processes. In *Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE'08)*, volume 5074 of *LNCS*, pages 32–47. Springer.

[Kumaran et al., 2003] Kumaran, S., Nandi, P., Heath, T., Bhaskaran, K., and Das, R. (2003). ADoc-Oriented Programming. In *Proceedings of the 2003 Symposium on Applications and the Internet (SAINT'03)*, pages 334–343. IEEE Computer Society.

[Küster, 2004] Küster, J. M. (2004). *Consistency Management of Object-Oriented Behavioral Models*. PhD thesis, University of Paderborn.

[Küster and Ryndina, 2007] Küster, J. M. and Ryndina, K. (2007). Improving Inconsistency Resolution with Side-effect Evaluation and Costs. In *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, volume 4735 of *LNCS*, pages 136–150. Springer.

[Küster et al., 2007] Küster, J. M., Ryndina, K., and Gall, H. (2007). Generation of Business Process Models for Object Life Cycle Compliance. In *Proceedings of the 5th International Conference on Business Process Management (BPM'07)*, volume 4714 of *LNCS*, pages 165–181. Springer.

[Küster and Stehr, 2003] Küster, J. M. and Stehr, J. (2003). Towards Explicit Behavioral Consistency Concepts in the UML. In *Proceedings of the 2nd International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (ICSE'03)*.

[Lassen and van der Aalst, 2006] Lassen, K. B. and van der Aalst, W. M. P. (2006). WorkflowNet2BPEL4WS: A Tool for Translating Unstructured Workflow Processes to Readable BPEL. In *Proceedings of the On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4275 of *LNCS*, pages 127–144. Springer.

[Leymann and Roller, 2000] Leymann, F. and Roller, D. (2000). *Production Workflow: Concepts and Techniques*. Prentice Hall PTR.

[Liu et al., 2007] Liu, R., Bhattacharya, K., and Wu, F. Y. (2007). Modeling Business Contexture and Behavior Using Business Artifacts. In *Proceedings of the 19th International Conference on Advanced Information Systems Engineering*, volume 4495 of *LNCS*, pages 324–339. Springer.

[Loos and Allweyer, 1998] Loos, P. and Allweyer, T. (1998). Object Orientation in Business Process Modeling through Applying Event Driven Process Chains (EPC) in UML. In *Proceedings of the 2nd International Enterprise Distributed Object Computing Workshop*, pages 102 – 112.

[Loos and Fettke, 2001] Loos, P. and Fettke, P. (2001). Towards an Integration of Business Process Modeling and Object-Oriented Software Development. In *Proceedings of the 5th International Symposium on Economic Informatics*, pages 835–843.

[Matoušek, 2003] Matoušek, P. (2003). *Verification of the Business Process Models*. PhD thesis, VŠB-Technical University of Ostrava.

[Mayer et al., 1992] Mayer, R. J., Painter, M. K., and deWitte, P. S. (1992). IDEF Family of Methods for Concurrent Engineering and Business Reengineering Applications. Technical report, Knowledge Based Systems Inc.

[McIlraith et al., 2001] McIlraith, S. A., Son, T. C., and Zeng, H. (2001). Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53.

[McMillan, 1993] McMillan, K. L. (1993). *Symbolic Model Checking*. Kluwer Academic Publishers.

[MDA, 2003] MDA (2003). MDA Guide Version 1.0.1, omg/2003-06-01. OMG Document.

[Mendling, 2007] Mendling, J. (2007). *Detection and Prediction of Errors in EPC Business Process Models*. PhD thesis, Vienna University of Economics and Business Administration, Vienna, Austria.

[Mendling et al., 2008] Mendling, J., Rosa, M. L., and ter Hofstede, A. H. M. (2008). Correcness of Business Process Models with Roles and Objects. Available online: `http://eprints.qut.edu.au/archive/00013845/`.

[Mens et al., 2006a] Mens, T., van der Staeten, R., and Warny, J.-F. (2006a). Graph-Based Tool Support to Improve Model Quality. In *Proceedings of the 1st Workshop on Quality in Modeling co-located with MoDELS 2006, Technical report 0627, Technische Universiteit Eindhoven*, pages 47–62.

[Mens and Van Der Straeten, 2006] Mens, T. and Van Der Straeten, R. (2006). Incremental Resolution of Model Inconsistencies. In *Proceedings of the 18th International Workshop on Recent Trends in Algebraic Development Techniques (WADT'06)*, volume 4409 of *LNCS*, pages 111–126. Springer.

[Mens et al., 2006b] Mens, T., Van Der Straeten, R., and D'Hondt, M. (2006b). Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)*, pages 200–214. Springer.

[Mertins and Jochem, 1999] Mertins, K. and Jochem, R. (1999). *Quality-Oriented Design of Business Processes*. Springer.

[Mitchell and Apt, 2001] Mitchell, J. C. and Apt, K. (2001). *Concepts in Programming Languages*. Cambridge University Press.

[MOF, 2002] MOF (2002). Meta Object Facility (MOF) Specification, Version 1.4, formal/02-04-03. OMG Available Specification.

[Monot, 2008] Monot, A. (2008). Object Life Cycle Explorer pour WebSphere Business Modeler. Final Year Internship Report, Ecole Nationale Supérieure des Mines de Nancy.

[Moore, 1956] Moore, E. F. (1956). Gedanken-Experiments on Sequential Machines. *Automata Studies,Annals of Mathematical Studies*, pages 129–153.

[Muccini, 2002] Muccini, H. (2002). An Approach for Detecting Implied Scenarios. In *Proceedings of the Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (ICSE'02)*.

[Muchnick, 1997] Muchnick, S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[Müller et al., 2006] Müller, D., Reichert, M., and Herbst, J. (2006). Flexibility of Data-Driven Process Structures. In *Business Process Management Workshops*, volume 4103 of *LNCS*, pages 181–192. Springer.

[Müller et al., 2007] Müller, D., Reichert, M., and Herbst, J. (2007). Data-Driven Modeling and Coordination of Large Process Structures. In *Proceedings of the 15th International Conference on Cooperative Information Systems (CoopIS'07)*, volume 4803 of *LNCS*, pages 131–149. Springer.

[Murata, 1989] Murata, T. (1989). Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77, pages 541–580.

[Nandi and Kumaran, 2005] Nandi, P. and Kumaran, S. (2005). Adaptive Business Object - A New Component Model for Business Integration. In *Proceedings of the 8th International Conference on Enterprise Information Systems*, pages 179–188.

[Nentwich et al., 2003] Nentwich, C., Emmerich, W., and Finkelstein, A. (2003). Consistency Management with Repair Actions. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 455–464. IEEE Computer Society.

[Nickel et al., 2000] Nickel, U. A., Niere, J., and Zündorf, A. (2000). Tool Demonstration: The FUJABA Environment. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 742–745. ACM Press.

[Nielson and Nielson, 1992] Nielson, H. R. and Nielson, F. (1992). *Semantics with Applications: A Formal Introduction*. Wiley.

[Nigam and Caswell, 2003] Nigam, A. and Caswell, N. (2003). Business Artifacts: An Approach to Operational Specification. *IBM Systems Journal*, 42(3):428–445.

[Nuseibeh and Easterbrook, 1999] Nuseibeh, B. and Easterbrook, S. (1999). The Process of Inconsistency Management: A Framework for Understanding. In *Proceedings of the 10th International Workshop on Database & Expert Systems Applications (DEXA'99)*, page 364. IEEE Computer Society.

[Nuseibeh et al., 2001] Nuseibeh, B., Easterbrook, S., and Russo, A. (2001). Making Inconsistency Respectable in Software Development. *Journal of Systems and Software*, 58(2):171–180.

[OCL, 2003] OCL (2003). UML 2.0 OCL Specification, ptc/03-10-14. OMG Document.

[Ouyang et al., 2006] Ouyang, C., Dumas, M., Breutel, S., and ter Hofstede, A. H. M. (2006). Translating Standard Process Models to BPEL. In *Proceedings of the 18th International Conference on Advanced Information Systems Engineering (CAiSE'06)*, volume 4001 of *LNCS*, pages 417–432. Springer.

[Peled, 1994] Peled, D. (1994). Combining Partial Order Reductions with On-the-fly Model-Checking. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV'94)*, pages 377–390. Springer.

[Peterson, 1981] Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*. Prentice-Hall.

[Pierce, 2002] Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press, Cambridge, MA, USA.

[Plotkin, 1981] Plotkin, G. D. (1981). A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, University of Aarhus.

[QVT, 2008] QVT (2008). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.0. OMG Specification.

[Rasch and Wehrheim, 2003] Rasch, H. and Wehrheim, H. (2003). Checking Consistency in UML Diagramms: Classes and State Machines. In *Proceedings of the 6th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'03)*, volume 2884 of *LNCS*, pages 229–243. Springer.

[Recker and Mendling, 2006] Recker, J. and Mendling, J. (2006). On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. In *CAiSE 2006 Workshop Proceedings - Eleventh International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD'06)*, pages 521–532.

[Redding et al., 2007] Redding, G., Dumas, M., ter Hofstede, A. H. M., and Iordachescu, A. (2007). Reconciling Object-oriented and Process-oriented Approaches to Information Systems Engineering. In *Proceedings of the 3rd International Workshop on Business Process Design (BPD'07)*.

[Reijers et al., 2003] Reijers, H. A., Limam, S., and van der Aalst, W. M. P. (2003). Product-Based Workflow Design. *Journal of Management Information Systems*, 20(1):229–262.

[Reijers and Vanderfeesten, 2004] Reijers, H. A. and Vanderfeesten, I. T. P. (2004). Cohesion and Coupling Metrics for Workflow Process Design. In *Proceedings of the 2nd International Conference on Business Process Management (BPM'04)*, volume 3080 of *LNCS*, pages 290–305. Springer.

[Russell et al., 2005] Russell, N., Hofstede, A., Edmond, D., and van der Aalst, W. M. P. (2005). Workflow Data Patterns: Identification, Representation and Tool Support. In *Proceedings of the 24th International Conference on Conceptual Modeling (ER'05)*, volume 3716 of *LNCS*, pages 353–368. Springer.

[Russell et al., 2004] Russell, N., ter Hofstede, A., Edmond, D., and van der Aalst, W. M. P. (2004). Workflow Data Patterns. Technical Report FIT-TR-2004-01, Queensland University of Technology, Brisbane.

[Russell et al., 2006a] Russell, N., ter Hofstede, A., van der Aalst, W. M. P., and Mulyar, N. (2006a). Workflow Control-Flow Patterns: A Revised View. Technical report, BPM Center Report BPM-06-22 , BPMcenter.org.

[Russell et al., 2006b] Russell, N., van der Aalst, W. M. P., ter Hofstede, A. H. M., and Wohed, P. (2006b). On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling. In *Proceedings of the 3rd Asia-Pacific Conference on Conceptual Modelling (APCCM'05)*, pages 95–104. Australian Computer Society.

[Ryndina et al., 2006] Ryndina, K., Küster, J. M., and Gall, H. (2006). Consistency of Business Process Models and Object Life Cycles. In *Workshops and Symposia at MoDELS 2006*, volume 4364 of *LNCS*, pages 80–90. Springer.

[Ryndina et al., 2007] Ryndina, K., Küster, J. M., and Gall, H. (2007). A Tool for Integrating Object Life Cycle and Business Process Modeling. In *Proceedings of the BPM Demonstration Program at the 5th International Conference on Business Process Management (BPM'07)*. CEUR-WS.

[Sadiq et al., 2004] Sadiq, S. W., Orlowska, M. E., Sadiq, W., and Foulger, C. (2004). Data Flow and Validation in Workflow Modelling. In *Proceedings of the 15th Australasian Database Conference*, volume 27 of *CRPIT*, pages 207–214. Australian Computer Society.

[Sadiq and Orlowska, 2000] Sadiq, W. and Orlowska, M. E. (2000). Analyzing Process Models Using Graph Reduction Techniques. *Information Systems*, 25(2):117–134.

[Sarstedt and Guttmann, 2007] Sarstedt, S. and Guttmann, W. (2007). An ASM Semantics of Token Flow in UML 2 Activity Diagrams. In *Perspectives of System Informatics: 6th International Andrei Ershov Memorial Conference (PSI'06)*, number 4378 in LNCS, pages 349–362. Springer.

[SCA, 2007] SCA (2007). SCA Service Component Architecture, Assembly Model Specification, SCA Version 1.00, Open SOA Collaboration Specification.

[Scheer, 2000] Scheer, A.-W. (2000). *Aris-Business Process Modeling*. Springer-Verlag New York, Inc.

[Schmidt, 2006] Schmidt, D. C. (2006). Model-Driven Engineering. *IEEE Computer*, 39(2):25.

[Schrefl and Stumptner, 2002] Schrefl, M. and Stumptner, M. (2002). Behavior-Consistent Specialization of Object Life Cycles. *ACM Transactions on Software Engineering and Methodology*, 11(1):92–148.

[Schürr, 1994] Schürr, A. (1994). Specification of Graph Translators with Triple Graph Grammars. In Mayr, E., Schmidt, G., and Tinhofer, G., editors, *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science (WG'94)*, pages 151–163. LNCS 903, Springer.

[Sendall and Kozaczynski, 2003] Sendall, S. and Kozaczynski, W. (2003). Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45.

[Sharp and McDermott, 2001] Sharp, A. and McDermott, P. (2001). *Workflow Modeling: Tools for Process Improvement and Application Development*. Artech House.

[Shlaer and Mellor, 1988] Shlaer, S. and Mellor, S. J. (1988). *Object-Oriented Systems Analysis: Modeling the World in Data*. Yourdon Press.

[Slonneger and Kurtz, 1995] Slonneger, K. and Kurtz, B. L. (1995). *Formal Syntax and Semantics of Programming Langauges: A Laboratory-Based Approach*. Addison-Wesley.

[Smith and Fingar, 2006] Smith, H. and Fingar, P. (2006). *Business Process Management: The Third Wave*. Meghan Kiffer Pr.

[Sommerville, 2006] Sommerville, I. (2006). *Software Engineering*. International Computer Science Series. Addison Wesley, 8th edition.

[Sommerville et al., 1998] Sommerville, I., Sawyer, P., and Viller, S. (1998). Viewpoints for Requirements Elicitation: A Practical Approach. In *Proceedings of the 3rd International Conference on Requirements Engineering (ICRE'98), Putting Requirements Engineering to Practice*, pages 74–81. IEEE Computer Society.

[Sommerville et al., 1999] Sommerville, I., Sawyer, P., and Viller, S. (1999). Managing Process Inconsistency Using Viewpoints. *IEEE Transactions on Software Engineering*, 25(6):784–799.

[Spanoudakis and Zisman, 2001] Spanoudakis, G. and Zisman, A. (2001). *Handbook of Software Engineering and Knowledge Engineering*, chapter Inconsistency Management in Software Engineering: Survey and Open Research Issues, pages 329–380. World Scientific Publishing Co.

[Sreedhar, 2006] Sreedhar, V. C. (2006). Data-Centric Security: Role Analysis and Role Typestates. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT'06)*, pages 170–179. ACM.

[Stoerrle, 2004] Stoerrle, H. (2004). Semantics of Control-Flow in UML 2.0 Activities. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*, pages 235–242. IEEE Computer Society.

[Stoerrle, 2005] Stoerrle, H. (2005). Semantics and Verification of Data Flow in UML 2.0 Activities. *Electronic Notes in Theoretical Computer Science*, 127(4):35–52.

[Stumptner and Schrefl, 2000] Stumptner, M. and Schrefl, M. (2000). Behavior Consistent Inheritance in UML. In *Proceedings of Conceptual Modeling - ER 2000*, volume 1920 of *LNCS*, pages 527–542. Springer.

[Taentzer, 2003] Taentzer, G. (2003). AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *AGTIVE*, volume 3062 of *LNCS*, pages 446–453.

[Taylor, 1911] Taylor, F. W. (1911). *The Principles of Scientific Management*. Harper and Brothers.

[Uchitel et al., 2001] Uchitel, S., Kramer, J., and Magee, J. (2001). Detecting Implied Scenarios in Message Sequence Chart Specifications. In *Proceedings of European Software Engineering Conference (ESEC/FSE'01)*.

[Uchitel et al., 2003] Uchitel, S., Kramer, J., and Magee, J. (2003). Synthesis of Behavioral Models from Scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115.

[Uchitel et al., 2004] Uchitel, S., Kramer, J., and Magee, J. (2004). Incremental Elaboration of Scenario-Based Specifications and Behavior Models Using Implied Scenarios. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 13(1):37–85.

[UML, 2007a] UML (2007a). OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2, formal/2007-11-02. OMG Specification.

[UML, 2007b] UML (2007b). OMG Unified Modeling Language (OMG UML), Version 2.1.2. OMG Specification.

[Valmari, 1992] Valmari, A. (1992). A Stubborn Attack on State Explosion. *Formal Methods in System Design*, 1(4):297–322.

[van der Aalst, 1999] van der Aalst, W. (1999). Formalization and Verification of Event-Driven Process Chains. *Information and Software Technology*, 41(10):639–650.

[van der Aalst, 1997] van der Aalst, W. M. P. (1997). Verification of Workflow Nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, volume 1248 of *LNCS*, pages 407–426. Springer.

[van der Aalst et al., 2001] van der Aalst, W. M. P., Barthelmess, P., Ellis, C. A., and Wainer, J. (2001). Proclets: A Framework for Lightweight Interacting Workflow Processes. *International Journal of Cooperative Information Systems*, 10(4):443–481.

[van der Aalst and Basten, 2001] van der Aalst, W. M. P. and Basten, T. (2001). Identifying Commonalities and Differences in Object Life Cycles using Behavioral Inheritance. In *Application and Theory of Petri Nets 2001*, volume 2075 of *LNCS*, pages 32–52. Springer.

[van der Aalst et al., 2002] van der Aalst, W. M. P., Hirnschall, A., and Verbeek, H. M. W. (2002). An Alternative Way to Analyze Workflow Graphs. In *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE'02)*, volume 2348 of *LNCS*, pages 535–552. Springer.

[van der Aalst et al., 2003] van der Aalst, W. M. P., ter Hofstede, A., Kiepuszewski, B., and Barros, A. (2003). Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51.

[van der Aalst and ter Hofstede, 2005] van der Aalst, W. M. P. and ter Hofstede, A. H. M. (2005). YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275.

[van der Aalst and van Hee, 2004] van der Aalst, W. M. P. and van Hee, K. (2004). *Workflow Management: Models, Methods, and Systems*. The MIT Press.

[van der Aalst et al., 2005] van der Aalst, W. M. P., Weske, M., and Grünbauer, D. (2005). Case Handling: A New Paradigm for Business Process Support. *Data & Knowledge Engineering*, 53(2):129–162.

[Van Der Straeten, 2005] Van Der Straeten, R. (2005). *Inconsistency Management in Model-Driven Engineering*. PhD thesis, Vrije Universiteit Brussel.

[Van Der Straeten and D'Hondt, 2006] Van Der Straeten, R. and D'Hondt, M. (2006). Model Refactorings through Rule-Based Inconsistency Resolution. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC'06)*, pages 1210–1217. ACM.

[Van Der Straeten et al., 2003] Van Der Straeten, R., Mens, T., Simmonds, J., and Jonckers, V. (2003). Using Description Logic to Maintain Consistency between UML Models. In *Proceedings of the 6th International Conference on The Unified Modeling Language, Modeling Languages and Applications (UML'03)*, volume 2863 of *LNCS*, pages 326–340. Springer.

[van Dongen et al., 2005] van Dongen, B. F., de Medeiros, A. K. A., Verbeek, H. M. W., Weijters, A. J. M. M., and van der Aalst, W. M. P. (2005). The ProM Framework: A New Era in Process Mining Tool Support. In *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets (ICATPN'05)*, volume 3536 of *LNCS*, pages 444–454. Springer.

[van Glabbeek, 1990] van Glabbeek, R. J. (1990). The Linear Time-Branching Time Spectrum (Extended Abstract). In *Proceedings of Theories of Concurrency: Unification and Extension (CONCUR'90)*, volume 458 of *LNCS*, pages 278–297. Springer.

[van Hee et al., 2006] van Hee, K., Sidorova, N., Somers, L., and Voorhoeve, M. (2006). Consistency in Model Integration. *Data Knowledge Engineering*, 56(1):4–22.

[Vanhatalo et al., 2008] Vanhatalo, J., Völzer, H., and Koehler, J. (2008). The Refined Process Structure Tree. In *Proceedings of the 6th International Conference on Business Process Management (BPM'08)*, volume 5240 of *LNCS*, pages 100–115. Springer.

[Vanhatalo et al., 2007] Vanhatalo, J., Völzer, H., and Leymann, F. (2007). Faster and More Focused Control-Flow Analysis for Business Process Models though SESE Decomposition. In *Proceedings of the 5th International Conference on Service-Oriented Computing (ICSOC'07)*, volume 4749 of *LNCS*, pages 43–55. Springer.

[Verbeek et al., 2001] Verbeek, H. M. W., Basten, T., and van der Aalst, W. M. P. (2001). Diagnosing Workflow Processes using Woflan. *The Computer Journal*, 44(4):246–279.

[Verlage, 1994] Verlage, M. (1994). Multi-View Modeling of Software Processes. In *Proceedings of the 3rd European Wokshop on Software Process Technology (EWSPT3)*, volume 772 of *LNCS*, pages 123–126. Springer.

[Vitolins and Kalnins, 2005] Vitolins, V. and Kalnins, A. (2005). Semantics of UML 2.0 Activity Diagram for Business Modeling by Means of Virtual Machine. In *Proceedings of the 9th IEEE International EDOC Enterprise Computing Conference (EDOC'05)*, pages 181–194. IEEE Computer Society.

[Wagner et al., 2003] Wagner, R., Giese, H., and Nickel, U. (2003). A Plug-In for Flexible and Incremental Consistency Management. In *Proceedings Workshop on Consistency Problems in UML-based Software Development*, Technical Report. Blekinge Institute of Technology, San Francisco.

[Wahler and Küster, 2008] Wahler, K. and Küster, J. M. (2008). Predicting Coupling of Object-Centric Business Process Implementations. In *Proceedings of the 6th International Conference on Business Process Management (BPM'08)*, volume 5240 of *LNCS*, pages 148–163. Springer.

[Wahler et al., 2008] Wahler, K., Küster, J. M., and Monot, A. (2008). Object Life Cycle Explorer for WebSphere Business Modeler. `http://www.alphaworks.ibm.com/tech/olcexplorer`.

[Wahler, 2008] Wahler, M. (2008). *Using Patterns to Develop Consistent Design Constraints*. PhD thesis, Swiss Federal Institute of Technology Zurich.

[Wang and Kumar, 2005] Wang, J. and Kumar, A. (2005). A Framework for Document-Driven Workflow Systems. In *Proceedings of the 3rd International Conference on Business Process Management (BPM'05)*, pages 285–301.

[Whittle and Schumann, 2000] Whittle, J. and Schumann, J. (2000). Generating Statechart Designs from Scenarios. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 314–323. ACM Press.

[Wohed et al., 2003] Wohed, P., van der Aalst, W. M. P., Dumas, M., and ter Hofstede, A. H. M. (2003). Analysis of Web Services Composition Languages: The Case of BPEL4WS. In *Proceedings of the 22nd International Conference on Conceptual Modeling (ER'03)*, volume 2813 of *LNCS*, pages 200–215. Springer.

[Wohed et al., 2005] Wohed, P., van der Aalst, W. M. P., Dumas, M., ter Hofstede, A. H. M., and Russell, N. (2005). Pattern-Based Analysis of the Control-Flow Perspective of UML Activity Diagrams. In *Proceedings of the 24th International Conference on Conceptual Modeling (ER'05)*, volume 3716 of *LNCS*, pages 63–78. Springer.

[Wohed et al., 2006] Wohed, P., van der Aalst, W. M. P., Dumas, M., ter Hofstede, A. H. M., and Russell, N. (2006). On the Suitability of BPMN for Business Process Modelling. In *Proceedings of the 4th International Conference on Business Process Management (BPM'06)*, volume 4102 of *LNCS*, pages 161–176. Springer.

[Wohlin et al., 1999] Wohlin, C., Runeson, P., and Höst, M. (1999). *Experimentation in Software Engineering: An Introduction*. International Series in Software Engineering. Springer, 1st edition.

[Wong and Gibbons, 2008] Wong, P. and Gibbons, J. (2008). A Process Semantics for BPMN. In *Proceedings of 10th International Conference on Formal Engineering Methods*, volume 5256 of *LNCS*. Springer.

[XPDL, 2008] XPDL (2008). Workflow Management Coalition Workflow Standard: Process Definition Interface XML Process Definition Language (XPDL), Version 2.1, WFMC-TC-1025-04-21-08. Workflow Management Coalition Specification.

[Yin, 2002] Yin, R. K. (2002). *Case Study Research: Design and Methods*, volume 5 of *Applied Social Research Methods Series*. Sage Publications, 3rd edition.

[zur Muehlen, 2004] zur Muehlen, M. (2004). *Workflow-Based Process Controlling. Foundation, Design, and Application of Workflow-Driven Process Information Systems*. Logos Verlag.

Appendix A

Pseudocode

Listing A.1: Action.producesLastStates() and getActionsProducingLastStates()

```
1 boolean producesLastStates()
2   for each (ObjectType t in this.getDataOutputs()) do
3     for each(State s in this.getProdStates(t)) do
4       if (t.getObjectLifeCycleModel().hasTransitionToFinalFrom(s))
5         return true;
6   return false;
7
8 Set getActionsProducingLastStates(Set actions)
9   Set result = new Set();
10  for each (Action act in actions) do
11    if (act.producesLastStates())
12      result.add(act);
13  return result;
```

Appendix B

Case Study Details

This appendix comprises the details about the identified validation errors, detected inconsistencies and applied resolutions in the CAD Management and IAA case studies. Tables B.1-B.3 are related to the CAD Management case study, while Tables B.4-B.9 to the IAA case study. The terms and abbreviations used in the tables are explained below.

- Contents of the Type column in Tables B.2, B.4-B.7:
  - NCnf_T: Non-Conformant Transition
  - NCnf_IT: Non-Conformant Initial Transition
  - NCnf_FT: Non-Conformant Final Transition
  - NCov_T: Non-Covered Transition
  - NCov_IT: Non-Covered Initial Transition
  - NCov_FT: Non-Covered Final Transition
- Contents of the Type column in Table B.1:
  - II: Invalid Transition
  - IIT: Invalid Initial Transition
  - IFT: Invalid Final Transition
  - ITS: Invalid Transition Split
  - MT: Missing Transition
  - O: Other
- Contents of the Source column in Table B.1:
  - CF: Control Flow (incorrect order of activities in a process model, stop or end nodes incorrectly connected in a process model, etc.)
  - DF: Data Flow (data incorrectly routed between activities, data inputs/outputs of an activity incorrectly modeled, data incorrectly routed through decisions, etc.)
  - SS: State Specification (input/output states for activities incorrectly modeled, states representing different decision outcomes are incorrectly modeled, etc.)
  - O: Other
- Contents of the Ranking column in Tables B.3 and B.9 give the number of the chosen resolution in the ranked list and the total number of resolutions in the list. The Effect column in these tables gives the number of inconsistencies removed as a side-effect and the number of inconsistencies induced as a side-effect.

Table B.1: Validation errors for CAD Management models

ID	Type	Transitions	Process model	Process elements	Source	Reason
Use case 1.1						
1	IIT	(Initial, Accepted)	UC1.1 STEP 2 Partner Design Activity To-Be	Verify import action	DF	Missing data flow between actions
2	IIT	(Initial, Rejected)	UC1.1 STEP 2 Partner Design Activity To-Be	Verify import action	DF	Missing data flow between actions
3	IFT	(Retrieved, Final)	UC1.1 STEP 2 Partner Design Activity To-Be	Check out workpackage for processing action	DF	Missing data flow between actions
4	IFT	(Identified, Final)	UC1.1 STEP 1 Design data exchange To-Be	Assign partner action	DF	Missing data flow to pass object to several actions
5	IIT	(Initial, Retrieved)	UC1.1 STEP 2 Partner Design Activity To-Be	Upload workpackage data action	DF	Missing data flow between actions
6	IFT	(Imported, Final)	UC1.1 STEP 2 Partner Design Activity To-Be	Verify import action	DF	Missing data flow between actions
7	IFT	(Accepted, Final)	UC1.1 STEP 2 Partner Design Activity To-Be	Partner design activities service	DF	Missing data flow between actions
8	ITS	Split Identified to Exported	UC1.1 STEP 1 Design data exchange To-Be	Send metadata action and Send design data action	DF/O	Two parallel actions represent Workpackage upload (meta and non-meta data), state of Workpackage changed in both
9	MT	(Accepted, Design_Completed)	UC1.1 STEP 2 Partner Design Activity To-Be	Partner design activities service	SS	Completion of partner design activities is not marked with state, although it's an important milestone
10	MT	(Design_Completed, Exported)	UC1.1 STEP 2 Partner Design Activity To-Be	Partner design activities service	SS	Completion of partner design activities is not marked with state, although it's an important milestone
11	MT	(Imported, Accepted)	UC1.1 STEP 2 Partner Design Activity To-Be	Verify import action	DF	Missing data flow between actions
12	MT	(Imported, Rejected)	UC1.1 STEP 2 Partner Design Activity To-Be	Verify action	DF	Missing data flow between actions
Use case 1.2						
1	IIT	(Initial, Exported)	UC1.2 STEP 2 Change actioned To-Be	Send confirmation of received information / automated replication action	DF	Missing data flow between actions

Table B.2: Inconsistencies for CAD Management models

ID	Type	States	Process elements	Process model	Object type
Use case 1.1					
1	NCnf_FT	Identified	-	Summary UC1 TO-BE	Workpackage
2	NCnf_FT	Retrieved	-	Summary UC1 TO-BE	Workpackage
3	NCnf_IT	Retrieved	Upload workpackage data	Summary UC1 TO-BE	Workpackage
4	NCnf_IT	Accepted	Verify import	Summary UC1 TO-BE	Workpackage
5	NCnf_IT	Rejected	Verify import	Summary UC1 TO-BE	Workpackage
6	NCnf_FT	Accepted	-	Summary UC1 TO-BE	Workpackage
7	NCnf_FT	Imported	-	Summary UC1 TO-BE	Workpackage
8	NCov_T	Imported, Rejected	-	Summary UC1 TO-BE	Workpackage
9	NCov_T	Accepted, Design_Completed	-	Summary UC1 TO-BE	Workpackage
10	NCov_T	Imported, Accepted	-	Summary UC1 TO-BE	Workpackage
11	NCov_T	Design_Completed, Exported	-	Summary UC1 TO-BE	Workpackage
Use case 1.2					
1	NCnf_IT	Exported	Send confirmation of received information / automated replication	Summary UC1.2 TO-BE	Workpackage

Table B.3: Resolution of inconsistencies in CAD Management models

ID	Auto/Side-effect/Manual	Auto			Manual	
		Type	Ranking	Effect	Description	Automatable
Use case 1.1						
1	Manual	-	-	-	Added data to a large set of edges	Yes
2	Auto	Add data and state to output edges	1/15	-2/0	-	-
3	Side-effect	-	-	-	-	-
4	Auto	Add data and state to input edges	1/12	-5/+1	-	-
5	Side-effect	-	-	-	-	-
6	Auto	Add data and state to output edges	1/14	-2/0	-	-
7	Side-effect	-	-	-	-	-
8	Side-effect	-	-	-	-	-
9	Side-effect	-	-	-	-	-
10	Side-effect	-	-	-	-	-
11	Side-effect	-	-	-	-	-
Inconsistencies introduced as side-effects						
12	Auto	Add data and state to input edges	1/15	-1/0	-	-
Use case 1.2						
1	Manual	-	-	-	Changed data types on several edges across process boundaries	No

Table B.4: Inconsistencies for Administer Claim process model and Claim object life cycle model (1/3)

ID	Type	States	Process elements	Process model	Object type	Reason
1	NCnf_T	Coverage_Confirmed, Granted	Decide On Claim	Administer Claim	Claim	This should be a transition from Under_Evaluation to Granted
2	NCnf_T	Rejected, Under_Evaluation	Allocate Claim To Adjuster action	Administer Claim	Claim	Rejected claims should first be closed and then they can be re-opened for re-evaluation; for this a Claim should go from Rejected, Closed, Open to Coverage_Evaluation
3	NCnf_T	Under_Evaluation, Rejected_Coverage_Not_Provided	Decide On Loss Coverage	Administer Claim	Claim	This should be a transition from Coverage_Evaluation to Rejected_Coverage_Not_Provided
4	NCnf_IT	Rejected	-	Administer Claim	Claim	State Notified should be the first state, but since it's not specified in the process model, process model analysis identified Rejected as the first state
5	NCnf_FT	Granted	Plan Service Fulfilment action	Settle Claim	Claim	Settle Claim contains an execution path through action Plan Service Fulfilment that leaves the Claim in state Granted without settling it; the Claim is not provided as a process output on this path as required
6	NCnf_T	Under_Evaluation, Coverage_Confirmed	Decide On Loss Coverage	Administer Claim	Claim	This should be a transition from Coverage_Evaluation to Coverage_Confirmed
7	NCnf_T	In_Dispute, Closed	Close Claim	Administer Claim	Claim	Due to incorrect state specification on the outgoing branches of the Benefit Offer Agreed decision in Settle Claim process model, a Claim can reach the Close Claim in state In_Dispute; according to the life cycle, In_Dispute claims should always go to Under_Negotiation state, from which they can be Rejected and then Closed or Granted, Settled and then Closed
8	NCnf_FT	Rejected	Claim Rejection Accepted decision	Administer Claim	Claim	If the Claim Rejection Accepted decision evaluates to true, the process ends and the Rejected claim is not closed
9	NCnf_T	Coverage_Confirmed, Rejected	Decide On Claim	Administer Claim	Claim	This should be a transition from Under_Evaluation to Rejected
10	NCnf_FT	Coverage_Confirmed	External Investigation Requirement decision	Record Claim Analysis	Claim	If the External Investigation Requirement decision evaluates to not required, the Claim is not passed on further in the process and therefore remains in state Coverage_Confirmed, which is identified as its last state

Table B.5: Inconsistencies for Administer Claim process model and Claim object life cycle model (2/3)

ID	Type	States	Process elements	Process model	Object type	Reason
11	NCov_T	Coverage_Confirmed, Under_Evaluation	-	Administer Claim	Claim	Under_Evaluation state is incorrectly used in the process model state specification; Claims should enter this intermediate state after the Coverage_Confirmed, which they currently do not
12	NCov_T	Coverage_Evaluation, Coverage_Confirmed	-	Administer Claim	Claim	Coverage_Evaluation state is not used at all in the process model, but Under_Evaluation is used in its place, which is incorrect according to the object life cycle model
13	NCov_T	Rejected, In_Dispute	-	Administer Claim	Claim	Rejected claims do not go through a dispute in the process model; only the Granted claims do
14	NCov_T	Coverage_Evaluation, Abandoned	-	Administer Claim	Claim	Abandonment of a claim is not represented at all in the process model, and also the Coverage_Evaluation state is not used
15	NCov_T	Settled, Open	-	Administer Claim	Claim	State Open is not represented at all in the process model
16	NCov_T	Granted, Under_Negotiation	-	Administer Claim	Claim	Granted claims can reach the state Under_Negotiation through state In_Dispute only in the process models, but not directly
17	NCov_T	Open, Coverage_Evaluation	-	Administer Claim	Claim	Open and Coverage_Evaluation states are not used in the process models
18	NCov_T	Nullified, Closed	-	Administer Claim	Claim	Nullification of claims is not represented in the process models
19	NCov_T	Under_Evaluation, Rejected	-	Administer Claim	Claim	Under_Evaluation state is used incorrectly in the process models and claims can only be rejected from state Coverage_Confirmed; there is a transition from Coverage_Confirmed to Under_Evaluation missing
20	NCov_T	Settled, Nullified	-	Administer Claim	Claim	Nullification of claims is not represented in the process models

Table B.6: Inconsistencies for Administer Claim process model and Claim object life cycle model (3/3)

ID	Type	States	Process elements	Process model	Object type	Reason
21	NCov_T	Under_Evaluation, Granted	-	Administer Claim	Claim	Under_Evaluation state is used incorrectly in the process models and claims can only be granted from state Coverage_Confirmed; there is a transition from Coverage_Confirmed to Under_Evaluation missing
22	NCov_T	Abandoned, Closed	-	Administer Claim	Claim	Abandonment of a claim is not represented at all in the process models
23	NCov_T	Notified, Open	-	Administer Claim	Claim	Notified and Open states are not used in the process models
24	NCov_IT	Notified	-	Administer Claim	Claim	Notified state is not used in the process models
25	NCov_T	Closed, Open	-	Administer Claim	Claim	Open state is not used in the process models; re-opening of Settled claims cannot be done in the process models
26	NCov_T	Under_Negotiation, Rejected	-	Administer Claim	Claim	State specification in the Settle Claim process model does not specify that claims go to state Rejected on the path where negotiation fails
27	NCov_T	Rejected, Closed	-	Administer Claim	Claim	Rejected claims are not closed in the process models
28	NCov_T	Coverage_Evaluation, Rejected_Coverage_Not_Provided	-	Administer Claim	Claim	Coverage_Evaluation state is not currently used in the process models, although it should be used to replace state Under_Evaluation
Inconsistencies introduced as side-effects						
29	NCnf_FT	Under_Evaluation	External Investigation Requirement decision	Record Claim Analysis	Claim	Side-effect of resolving inconsistency 1
30	NCnf_IT	Closed	-	Administer Claim	Claim	Side-effect of resolving inconsistency 2
31	NCnf_T	Closed, Coverage_Evaluation	Allocate Claim To Adjuster	Validate Claim	Claim	Side-effect of resolving inconsistency 3
32	NCnf_FT	Settled	Request Provider Service	Settle Claim	Claim	Side-effect of resolving inconsistency 5

Table B.7: Inconsistencies for Administer Claim process model and Benefit In Claim object life cycle model

ID	Type	States	Process elements	Process model	Object type
1	NCnf_T	Offered, Requested	Prepare Claim Discharge action	Administer Claim	Benefit In Claim
2	NCnf_FT	Requested	Administer Claim action	Administer Claim	Benefit In Claim
3	NCnf_FT	Rejected	Benefit Offer Agreed action	Settle Claim	Benefit In Claim
4	NCnf_FT	Requested	Record Benefit Payment action	Settle Claim	Benefit In Claim
5	NCnf_FT	Requested	Plan Service Fulfilment action	Settle Claim	Benefit In Claim
6	NCov_T	Offered, Obsolete	-	Administer Claim	Benefit In Claim
7	NCov_T	Settle, Nullified	-	Administer Claim	Benefit In Claim
8	NCov_T	Accepted, Settled	-	Administer Claim	Benefit In Claim
9	NCov_T	Draft, Rejected	-	Administer Claim	Benefit In Claim
10	NCov_IT	Requested	-	Administer Claim	Benefit In Claim
11	NCov_T	Requested, Obsolete	-	Administer Claim	Benefit In Claim
12	NCov_T	Requested, Rejected	-	Administer Claim	Benefit In Claim
13	NCov_T	Requested, Granted	-	Administer Claim	Benefit In Claim
14	NCov_FT	Nullified	-	Administer Claim	Benefit In Claim
15	NCov_T	Granted, Settled	-	Administer Claim	Benefit In Claim
16	NCov_T	Offered, Accepted	-	Administer Claim	Benefit In Claim
16	NCov_FT	Settled	-	Administer Claim	Benefit In Claim

Table B.8: Resolution of inconsistencies between Administer Claim process model and Claim object life cycle model

ID	Auto/Side-effect/Manual	Auto			Manual	
		Type	Ranking	Effect	Description	Automatable
1	Manual	-	-	-	Assigned Under_Evaluation as the output state of Determine Investigation Requirements action in Administer Claim process model	No
2	Manual	-	-	-	Created a path from Review Claim Rejection action to Close Claim action and from Close Claim to Provide Additional Data and Record Claim action if rejection of the claim is not accepted	No
3	Manual	-	-	-	Changed output state of Allocate Claim To Adjuster action in Validate Claim process model from Under_Evaluation to Coverage_Evaluation	Yes (partially)
4	Side-effect	-	-	-	-	-
5	Auto	Add data and state to output control edge	1/27	-1/0	-	-
6	Side-effect	-	-	-	-	-
7	Manual	-	-	-	Changed state of Claim on outgoing branch of Benefit Offer Agreed decision in Settle Claim process model labeled Offer Not Agreed from In_Dispute to Under_Negotiation and added a Negotiate Benefit Offer action succeeded by a decision to change the state of Claim to either Granted or Rejected	No
8	Side-effect	-	-	-	-	-
9	Side-effect	-	-	-	-	-
10	Side-effect	-	-	-	-	-
11	Side-effect	-	-	-	-	-
12	Side-effect	-	-	-	-	-
13	Manual	-	-	-	Transition from Rejected to In_Dispute removed from the object life cycle model	Yes
14	Manual	-	-	-	Added a Claim Abandoned decision followed by a Record Claim Abandonment action after the Allocate Claim To Adjuster action in Validate Claim process model, as well as control and data flows in the Administer Claim process model to Abandoned Claims	Yes (partially)

Table B.9: Resolution of inconsistencies

ID	Auto/Side-effect/Manual	Auto			Manual	
		Type	**Ranking**	**Effect**	**Description**	**Automatable**
15	Manual	-	-	-	Transition from Settled to Open removed from the object life cycle model	Yes
16	Side-effect	-	-	-	-	-
17	Manual	-	-	-	Added a Set Claim State To Open action to follow the output branch of the Information Completed decision labeled Completed in Record Claim process model	No
18	Manual	-	-	-	Added a Claim To Be Nullified? decision after the Benefit Settled decision branch labeled Settled in the Administer Claim process model and a Nullify Claim action, connecting it to Close Claim action	Yes (partially)
19	Side-effect	-	-	-	-	-
20	Side-effect	-	-	-	-	-
21	Side-effect	-	-	-	-	-
22	Side-effect	-	-	-	-	-
23	Manual	-	-	-	Added state Notified to the outgoing edge of Notify Claim	Yes
24	Side-effect	-	-	-	-	-
25	Side-effect	-	-	-	-	-
26	Side-effect	-	-	-	-	-
27	Side-effect	-	-	-	-	-
28	Side-effect	-	-	-	-	-
Inconsistencies introduced as side-effects						
29	Manual	-	-	-	Connected the unconnected Claim pin in the outgoing branch of External Investigation Requirement decision to the incoming branch of the merge in Record Claim Analysis process model	No
30	Side-effect	-	-	-	-	-
31	Side-effect	-	-	-	-	-
32	Manual	-	-	-	Added a Claim output to Request Provider Service action and connected it to the output of the Settle Claim process model	No

List of Figures

List of Tables

Index

Curriculum Vitae

Personal Information

Ksenia Wahler (born Ryndina)

Date of Birth: July 15, 1980 (Moscow, Russia)

Nationality: Russian, South African

Academic Record

April 2005 - March 2009

University of Zurich, Switzerland
Doctor in Informatics, advisor Prof. Dr. Harald C. Gall

February 1999 - March 2005

University of Cape Town, South Africa
Master of Science in Computer Science, advisor Prof. Dr. Pieter Kritzinger
Bachelor of Business Science with Computer Science major

Relevant Work Experience

October 2004 - February 2009

IBM Zurich Research Laboratory, Switzerland
Pre-doctoral researcher in the Business Integration Technologies research group
Intern for 6 months before starting doctoral studies

February 2001 - September 2004

University of Cape Town, South Africa
Teaching assistant and tutor in the Computer Science department
Assistant to sub-editor of the SAIEE Transactions journal, Special Issue on Software Engineering and Formal Methods
Assistant researcher in the Data Network Architectures research group